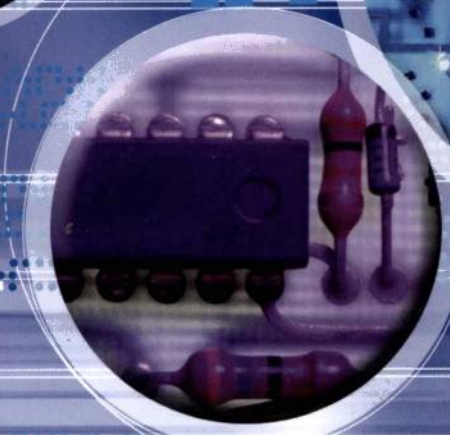
NEW WCDP

例说51单片机（C语言版）

张义和　王敏男　许宏昌　余春长　编著
袁彩霞　改编







人民邮电出版社
POSTS & TELECOM PRESS

NEW WCDP

例说51单片机（C语言版）



封面设计：胡平利

分类建议：计算机/硬件技术/单片机

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-17326-3

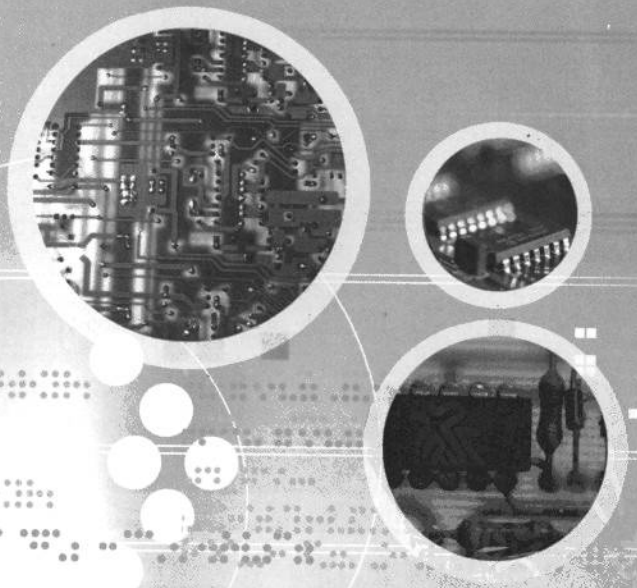
9 787115 173263 >

ISBN 978-7-115-17326-3/TP

定价：55.00 元

例说51单片机（C语言版）

张义和　王敏男　许宏昌　余春长　编著
袁彩霞　改编



人民邮电出版社
北京

图书在版编目（CIP）数据

例说 51 单片机：C 语言版/张义和等编著；袁彩霞改编.
北京：人民邮电出版社，2008.4
ISBN 978-7-115-17326-3

Ⅰ. 例… Ⅱ. ①张…②袁… Ⅲ. ①单片微型计算机—程序设计②C 语言—程序设计 Ⅳ. TP368.1 TP312

中国版本图书馆 CIP 数据核字（2007）第 193340 号

内 容 提 要

本书通过众多实例向读者介绍了51单片机的各种功能。全书分为14章，分别介绍了51单片机基础知识、μVision 3和Keil C开发环境、输入口及输出口的应用、中断的应用、定时器及计数器的应用、串行口的应用、声音的产生、步进马达的控制、ADC和DAC的应用、LED阵列的应用、LCD模块的应用、直流马达的控制等内容。附录中还对常用的在线仿真器进行了介绍，并提供了开发中常用的中英文名词对照表。

本书在写法上非常独特。每一章中，在简单介绍相关知识后，马上辅以内容详细、针对性强的“实例演练”，每个实例都有详细的零件介绍、电路图、程序流程图以及程序代码。在程序中，每条指令都提供了注解，让读者能够快速理解程序的内涵，并能够以最快的方式上手。其后又提出了具有启发性的问题，可以帮助读者巩固所学知识，达到举一反三的目的。

本书适合单片机硬件爱好者阅读。另外，本书还可作为各大中专院校相关专业的教学参考书。

例说 51 单片机（C 语言版）

◆ 编　　著　张义和　王敏男　许宏昌　余春长
改　　编　袁彩霞
责任编辑　翟　磊

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
印张：32.75
字数：759 千字　　2008 年 4 月第 1 版
印数：1-5 000 册　　2008 年 4 月北京第 1 次印刷
著作权合同登记号　图字：01-2008-0909
ISBN 978-7-115-17326-3/TP

定价：55.00 元

读者服务热线：(010)67132705　印装质量热线：(010)67129223
反盗版热线：(010)67171154

前　言

例说　单片机，单片机，还是单片机

从8048开始，“单片机”开始受到人们的欢迎，8051成为了机电、电子专业院校的必修课程。当然，应用单片机制作系统，既简单又经济实惠，早已成为学生们的最爱！在电路设计方面，对于需求量较大、电路较复杂者，可采用客户指定设计的IC（即ASIC），例如芯片组（chip set）、控制芯片等，虽然ASIC的单位成本不高，但总体成本很高、开发时间也比较长；对于需求量不是很大，而电路变化较多者，则采用可编程器件（如CPLD、FPGA）。不过CPLD、FPGA的单价较高。由此可见，目前对于中、小量的电路设计，还是采用单片机最合适。

例说　开辟优质牧场

在过去十多年里，笔者一直从事电路设计（OrCAD、Protel）与单片机的教学，带领学生从实体电路的设计、单片机程序设计，直到完成整个专题。笔者采用过不少的教材，而市面上有关8051的书籍虽多，但其结构类似，缺乏完整而适合教学的教材。多年前，笔者打破“为喝一杯鲜奶，养一头母牛”的观念，编写了第一本较适用的单片机教材；而现在更觉得需要开辟一个优质的牧场，于是以Keil C与Atmel 89S51为基础，撰写了这本优质的51单片机教材。

例说　一本非常适合教学的单片机教材

“让教与学更有效率”是例说系列图书的宗旨，本书的编排具有下列特色。

采用适合于教学的编排与内容

按教学的需要与顺序，将89S51的硬件、外围电路与器件，分布于各章节之中，以降低学习的难度，提升学习的兴趣。每个单元都有许多对应的实例与练习，每个实例与练习里都有详细的器件介绍与设计数据、电路图、程序流程图以及程序设计。在程序之中，每行指令右边都提出了实时的解释，让读者能快速明白程序的意义，而不必翻来翻去的。在实例演练之后，还提出了具有启发性的问题，让读者能够做到举一反三。

精细的美编与绘图

本书版式设计精美，图文并茂，芯片的图示由作者精心绘制而成，比厂商PDF文档中的图示更清晰，更有助于学习。

严谨的编校与教学辅助

本书特邀请多位具有丰富的8051教学经验的老师及产业界的好友帮忙与指导。特别是王敏男老师，本书里的程序由他负责89S51在线烧录，实验板则是由李国彰工程师操刀的！而张显盛老师、许宏昌老师、余春长副总等，不但协助校阅稿件，还提供了不少好主意！

内容精彩实用

笔者深知，市面上不乏内容丰富、程序功力深厚的8051相关书籍，但笔者更有把握的是，本书更“好看”！让读者看得舒服、看得清楚，看到该看的部分、看到想看的部分，在不知不觉之中成为单片机的高手。另外，在此特别感谢新华计算机、力浦电子及长高科技等公司提供的帮助，为本书增色不少！也期待读者不吝指正。

张义和敬上

目 录

第1章 轻松看51单片机

1.1 微型计算机与单片机 ……2
1.2 8051基础知识 ……2
1.2.1 8051的结构 ……3
1.2.2 89C51/89S51的封装与引脚 ……4
1.2.3 8051的基本电路 ……8
1.2.4 MCS-51系列 ……11
1.2.5 关于Atmel的51系列 ……12
1.3 了解MCS-51的存储器结构 ……13
1.3.1 程序存储器 ……14
1.3.2 数据存储器 ……14
1.3.3 特殊功能寄存器 ……17
1.4 8051的时序分析与复位 ……21
1.4.1 时序分析 ……21
1.4.2 复位 ……23
1.5 89S51的开发流程与工具 ……23
1.5.1 传统开发工具 ……24
1.5.2 Altium Designer电路设计软件 ……26
1.5.3 Keil μVision 3整合开发环境 ……27
1.5.4 89S51的在线烧录功能 ……28
1.6 实例演练 ……28
1.7 实时练习 ……34

第2章 认识μVision 3与Keil C

2.1 μVision 3s环境简介 ……38
2.1.1 认识μVision 3环境 ……38
2.1.2 项目管理与选项 ……41
2.1.3 认识调试/仿真环境 ……43
2.1.4 外围操作 ……44
2.2 Keil C语言的基本结构 ……48
2.3 变量、常数与数据类型 ……52
2.3.1 数据类型 ……53
2.3.2 变量名称与保留字 ……55
2.3.3 变量的作用范围 ……56
2.4 存储器形式与模式 ……57
2.4.1 存储器形式 ……57
2.4.2 存储器模式 ……58
2.5 Keil C的运算符 ……59
2.6 Keil C的流程控制 ……64
2.6.1 循环指令 ……65
2.6.2 选择指令 ……68
2.6.3 跳转指令 ……70
2.7 数组与指针 ……71
2.7.1 数组 ……71
2.7.2 指针 ……72
2.8 函数与中断子程序 ……73
2.8.1 函数 ……73
2.8.2 中断子程序 ……73
2.9 Keil C的预处理命令 ……74
2.10 实时练习 ……75

第3章 输出口的应用

3.1 认识8051的输入/输出口 ……78

3.2 输出电路设计 ……80
3.2.1 驱动LED ……80
3.2.2 驱动蜂鸣器 ……82
3.2.3 驱动继电器 ……82
3.2.4 7段LED数码管 ……85
3.3 实例演练 ……88
3.3.1 驱动蜂鸣器实验 ……88
3.3.2 驱动继电器实验 ……90
3.3.3 霹雳灯实例演练 ……93
3.3.4 驱动7段LED数码管实验 ……96
3.4 实时练习 ……98

第4章 输入口的应用

4.1 认识输入口 ……102
4.2 输入设备与输入电路设计 ……102
4.2.1 输入设备 ……102
4.2.2 输入电路设计 ……106
4.2.3 抖动与防抖动 ……107
4.3 实例演练 ……110
4.3.1 指拨开关控制实例演练 ……111
4.3.2 按钮ON-OFF控制实例演练 ……112
4.3.3 按钮切换式控制实例演练 ……114
4.3.4 按钮开关应用实例演练 ……117
4.3.5 BCD数字型指拨开关实例演练 ……119
4.3.6 多重按钮开关实例演练 ……121
4.3.7 按钮开关放开后动作实例演练 ……128
4.4 实时练习 ……130

第5章 输入/输出口的高级应用

5.1 键盘扫描 ……134

5.1.1 键盘扫描原理 ……136
5.1.2 4×4键盘扫描程序解析 ……139
5.1.3 认识MM74C922/MM74C923 ……142
5.2 7段LED数码管扫描 ……144
5.2.1 认识7段LED数码管模块 ……144
5.2.2 认识7447/7448 ……147
5.2.3 认识74138/74139 ……151
5.3 静态显示与动态显示 ……153
5.3.1 使用BCD译码器 ……153
5.3.2 直接驱动 ……156
5.3.3 使用扫描译码器 ……157
5.3.4 闪烁 ……158
5.3.5 交替显示 ……159
5.3.6 飞入 ……160
5.3.7 跑马灯 ……161
5.4 同时扫描 ……162
5.5 实例演练 ……165
5.5.1 四位数7段LED数码管实例演练 ……165
5.5.2 直接驱动7段LED数码管实例演练 ……169
5.5.3 跑马灯实例演练 ……172
5.5.4 4×4键盘与7段LED数码管实例演练 ……174
5.5.5 MM74C922实例演练 ……178
5.6 实时练习 ……179

第6章 中断的应用

6.1 认识89S51的中断 ……182
6.1.1 MCS-51的中断 ……182
6.1.2 中断启用寄存器IE ……183
6.1.3 中断优先级寄存器IP ……184
6.1.4 定时器/计数器控制寄存器TCON ……185
6.1.5 中断向量 ……186

6.1.6 中断的应用 ……186
6.2 中断子程序的仿真 ……187
6.3 实例演练 ……188
6.3.1 外部中断INT0实例演练 ……188
6.3.2 外部中断INT1实例演练 ……191
6.3.3 两个外部中断实例演练 ……194
6.3.4 键盘中断实例演练 ……198
6.4 实时练习 ……200

第7章 定时器/计数器的应用

7.1 8051的定时器/计数器 ……204
7.1.1 MCS-51的定时器/计数器 ……204
7.1.2 定时器/计数器模式寄存器TMOD ……204
7.1.3 定时器/计数器控制寄存器TCON ……207
7.1.4 计数寄存器 ……208
7.1.5 定时器/计数器的应用 ……209
7.2 8052的Timer 2 ……210
7.2.1 T2CON寄存器 ……210
7.2.2 捕获模式 ……212
7.2.3 自动加载模式 ……212
7.2.4 比特率产生器模式 ……213
7.3 8051的省电模式 ……214
7.3.1 闲置模式 ……215
7.3.2 功率下降模式 ……215
7.3.3 电源控制寄存器PCON ……216
7.4 认识看门狗定时器 ……216
7.5 实例演练 ……218
7.5.1 闪烁灯实例演练 – 查询方式 ……218
7.5.2 闪烁灯实例演练 – 中断方式 ……221
7.5.3 60s定时器实例演练 ……223
7.5.4 码表实例演练 ……226

7.5.5 频率发生器实例演练 ……230
7.5.6 计频器实例演练 ……234
7.5.7 看门狗定时器实例演练 ……238
7.6 实时练习 ……241

第8章 串行口的应用

8.1 数据传输概念 ……244
8.2 认识74164/74165 ……245
8.3 认识MCS-51的串行口 ……247
8.4 认识SCON串行口控制寄存器 ……249
8.5 比特率设定 ……251
8.6 认识MAX232 ……252
8.7 多处理器间的数据传输 ……253
8.8 实例演练 ……254
8.8.1 移位式数据串入实例演练 ……254
8.8.2 移位式数据串出实例演练 ……256
8.8.3 mode 1实例演练 ……258
8.8.4 mode 2实例演练 ……260
8.8.5 mode 3实例演练 ……261
8.8.6 8051 vs 8051实例演练 ……263
8.9 实时练习 ……264

第9章 声音的产生

9.1 发声电路 ……268
9.2 音调与节拍 ……269
9.3 音调的产生 ……271
9.4 节拍的产生 ……277
9.5 实例演练 ……284
9.5.1 简易电子琴实例演练 ……285
9.5.2 DoReMi实例演练 ……288

9.5.3 生日快乐歌实例演练 ……290
9.5.4 快乐点唱机实例演练一 ……293
9.5.5 快乐点唱机实例演练二 ……299
9.6 实时练习 ……302

第10章 步进马达的控制

10.1 认识步进马达 ……306
10.1.1 步进马达的结构 ……306
10.1.2 步进马达的动作 ……307
10.1.3 步进马达的定位 ……310
10.2 步进马达驱动电路 ……312
10.2.1 小型步进马达的驱动电路 ……312
10.2.2 达林顿电晶体管驱动电路 ……313
10.2.3 FT5754驱动电路 ……314
10.3 实例演练 ……315
10.3.1 以延迟函数产生驱动信号实例演练 ……315
10.3.2 以定时器产生驱动信号实例演练 ……317
10.3.3 1-2相驱动实例演练 ……320
10.3.4 方向控制实例演练 ……321
10.4 实时练习 ……324

第11章 ADC与DAC的应用

11.1 模拟-数字转换原理 ……328
11.2 认识AD转换IC ……331
11.3 数字-模拟转换原理 ……336
11.4 认识DA转换IC ……343
11.5 内含ADC的51系列 ……347
11.6 认识温度传感器 ……349
11.7 实例演练 ……350
11.7.1 ADC连续转换实例演练 ……350

11.7.2 ADC交互式转换实例演练之一 ……354
11.7.3 ADC交互式转换实例演练之二 ……357
11.7.4 ADC的温控实例演练 ……359
11.7.5 DAC实例演练 ……362
11.8 实时练习 ……364

第12章 LED阵列的应用

12.1 认识LED阵列 ……368
12.2 LED阵列驱动电路 ……372
12.3 LED阵列显示方式 ……376
12.4 LED阵列动态显示 ……378
12.4.1 水平移动 ……378
12.4.2 垂直移动 ……386
12.5 实例演练 ……389
12.5.1 8×8 LED阵列静态展示实例演练……389
12.5.2 8×8 LED阵列多色显示……392
12.5.3 8×8 LED阵列平移实例演练……395
12.5.4 8×8 LED阵列跑马灯实例演练……399
12.5.5 8×8 LED阵列垂直移动实例演练……402
12.5.6 8×8 LED阵列卷动实例演练……406
12.5.7 16×16 LED阵列显示实例演练……408
12.6 实时练习 ……412

第13章 LCD模块的应用

13.1 认识LCD模块 ……416
13.2 中文LCD模块 ……421
13.3 LCM控制指令 ……424
13.4 LCM的初始设定与常用函数 ……428
13.5 LCM与8051的连接 ……437
13.6 实例演练 ……437

13.6.1 LCD文字显示实例演练 ……438
13.6.2 自编字型图案实例演练 ……441
13.7 实时练习 ……448

第14章 直流马达的控制

14.1 认识直流马达 ……452
14.2 直流马达的驱动方式 ……453
14.3 直流马达的PWM控制 ……461
14.4 实例演练 ……462
14.4.1 继电器开关控制实例演练 ……463
14.4.2 继电器的方向控制实例演练 ……465
14.4.3 桥式方向控制实例演练 ……467
14.4.4 PWM控制实例演练 ……469
14.5 实时练习 ……472

附录A 开发工具简介

A.1 WINICE-51/52E在线仿真器 ……476
A.2 WICE-52在线仿真器 ……487
A.3 PICE-52在线仿真器……491
A.4 89S51在线烧录实验板……496

附录B 中英文名词对照表

第1章 轻松看51单片机

本章主要包括以下三部分。

8051部分：

8051的基础知识，包括结构、引脚、封装、MCS-51系列与基本电路等。

8051的结构，包括存储器配置、时序分析等。

开发工具部分：

8051软硬件的开发流程，包括源程序的编写、编译、链接以及软硬件仿真等。

程序与实践部分：

高低四位交替闪烁灯的程序设计及其编译、链接与软件仿真。

1.1 微型计算机与单片机

一般来说，微型计算机包括中央处理器（CPU）、存储器（Memory）及输入/输出单元（I/O）三大部分，如图1-1所示。CPU就像是人的大脑一样，控制整个系统的运行；存储器则存放系统运行所需要的程序及数据，包括只读存储器（**Read Only Memory**，简称**ROM**）和随机访问存储器（**Random Access Memory**，简称**RAM**）。通常ROM用来储存程序或永久性的数据，称为程序存储器，RAM则用来储存程序执行时的临时数据，称为数据存储器；I/O是微型计算机与外部沟通的通道，其中包括输出口与输入口。这三部分分别由不同的部件（IC）组成，把它们组装在电路板上即可以形成一个微型计算机系统。

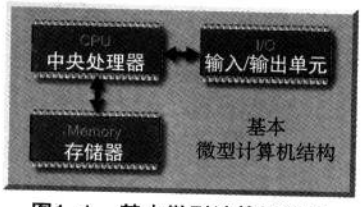


图1-1 基本微型计算机结构

单片机微控制器就是把中央处理器、存储器、输入/输出端口等全部放置在一个芯片里，如图1-2所示，只要再配置几个小器件，如电阻、电容、石英晶体、连接器等，即成为完整的微型计算机。因此单片机整个系统的体积小、成本低、可靠度高，是目前微机控制系统的主流产品。

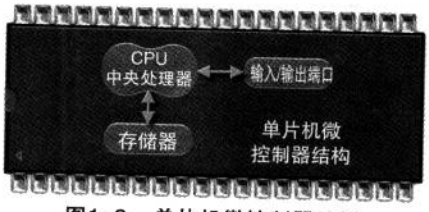


图1-2 单片机微控制器结构

由于微型计算机的主要功能是控制，因此，在单片机里，不太在乎其存储器大小、位数，而强调其输入/输出功能。

1.2 8051基础知识

“**8051**”源自Intel公司的MCS-51系列芯片，但目前所采用的8051并不限于Intel公司所生产的芯片，而以其他厂商所发行的兼容芯片为主，如Atmel公司的89C51/89S51系列，

其价格便宜、质量稳定、开发工具齐全，受到学校和培训机构的普遍欢迎。

在此先介绍8051的基础知识，包括基本结构、引脚、基本电路及51系列等。其中很多资料最好要“记”在脑里，笔者也会提供许多快速背记的技巧，让读者能在极短的时间里记住40个引脚、基本电路等。

1.2.1 8051的结构

8051单片机发展至今，虽然有许多厂商各自开发不同的兼容芯片，但其基本结构并没有多大的变动，标准的8051结构如图1-3所示。

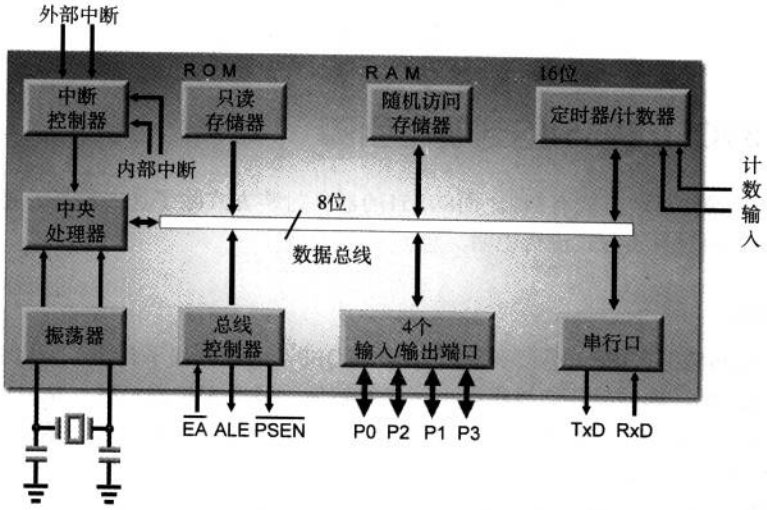


图1-3 MCS-51内部基本结构图

- 8051为8位微控制器[1]。
- 程序存储器ROM：内部有4KB、外部最多可扩展至64KB。
- 数据存储器RAM：内部有128B、外部最多可扩展至64KB。
- 4组可位寻址[2]的8位输入/输出端口，即P0、P1、P2及P3。
- 1个全双工串行口，即UART；两个16位定时器/计数器。
- 5个中断源，即INT0、INT1、T0、T1、TXD/RXD。
- 111个指令码。

[1] 8位指的是微控制器内部数据总线或寄存器一次处理数据的位数。相对于目前个人计算机（PC）所用的CPU：早期的CPU从 8088/8086到80286都属于16位CPU，而从80386到Pentium 3都属于32位的CPU。尽管如此，目前所采用的单片机仍是以8位为主，只有在特殊场合才会采用16位的单片机，如8096等。

[2]通常存储器的操作是以字节为单位，“可位寻址”是指访问存储器、寄存器或输入/输出端口时，可指定其中的一个位，例如要指定P0输入/输出端口中的bit 1，则指定为P0.1即可，如图1-4所示。

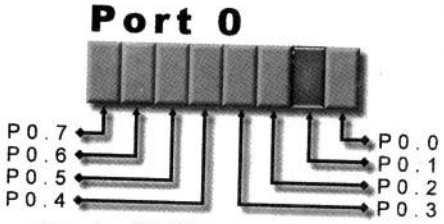


图1-4 批定输入/输出端口中的bit 1

1.2.2 89C51/89S51的封装与引脚

89C51的器件封装方式有3种。而89S51的器件封装方式除了这3种外，还有PDIP42器件封装。下面对封装方式进行说明。

QFP封装

89C51/89S51的PQFP（小型平面封装）或TQFP（薄四方扁平封装）为扁平的44个引脚表面贴式封装，这种封装的体积很小、成本较低，适合于机器粘贴，是目前主流的封装方式，但在学校或培训机构里，这种封装并不适用。如图1-5所示，在俯视图中，左上方有个记号的脚为第1脚，然后逆时针排序，分别为第2、3…44脚，其中包括3个空脚，相邻两个脚的间距为0.8mm，器件厚度（高度）为1.2mm。

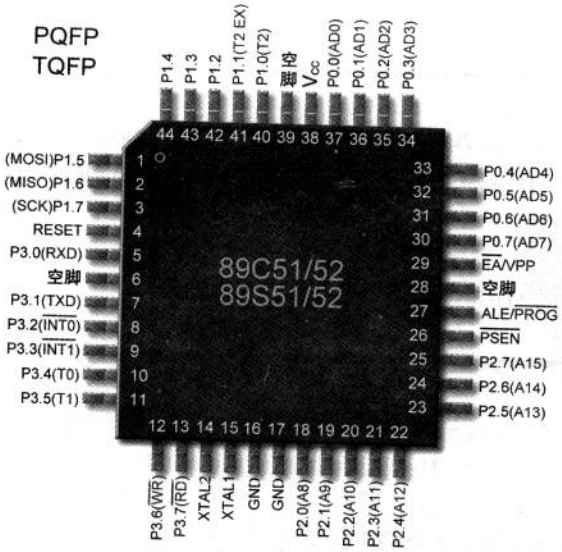


图1-5 PQFP（TQFP）封装

PLCC封装

PLCC（有引线塑料芯片载体封装）也是89C51/89S51常用的封装方式，这也是44个表面贴式引脚（SMT）的封装，其中包括4个空脚，而其引脚编号与QFP封装非常类似（相容）。如图1-6所示，在俯视图里，上面中间有个记号者为第1脚，然后逆时针排序，分别为第2、3…44脚，相邻两个脚的间距为1.270mm，器件高度（含引脚）为4.572mm。

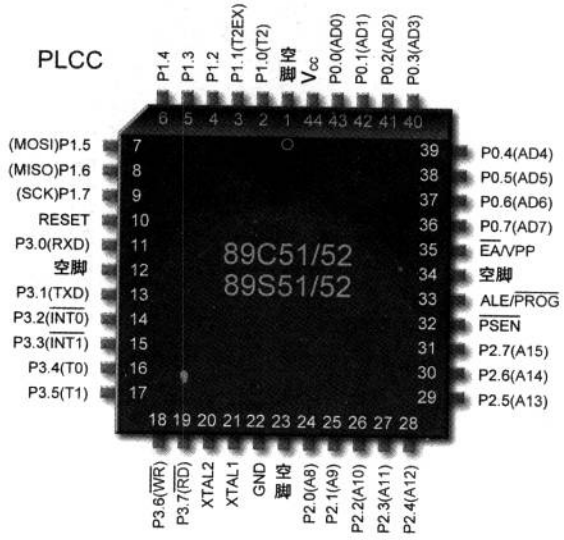


图1-6 PLCC封装

一般来说，这是一种表面贴式的器件，可直接贴于电路板上，而不必钻孔（其引脚如图1-7所示）。在研发、实验或教学时，还可利用底座，这样可以缩短开发与生产的差距。

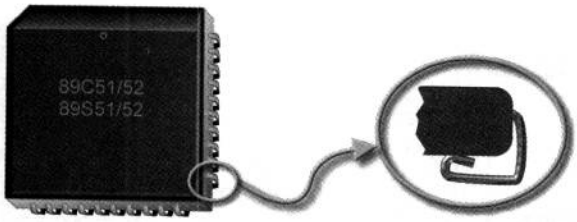


图1-7 PLCC引脚

双列直插式封装的PDIP42

89S51的针脚式封装有两种，第一种42个引脚双并排的PDIP（DIP双列直插封装），也称为PDIP42。第二种40个引脚双并排封装，与89C51、MCS-51兼容，稍后将对其进行说明。如图1-8所示，在双并排封装里，俯视图左上方有个记号者为第1脚，然后逆时针排序，分别为第2、3…42脚。相邻两个脚的间距为1.588mm，器件长度为36.96mm，而两排引脚的间距为13.97mm，器件厚度为4.826mm（不含引脚），与一般的面包板或IC底座不符合。

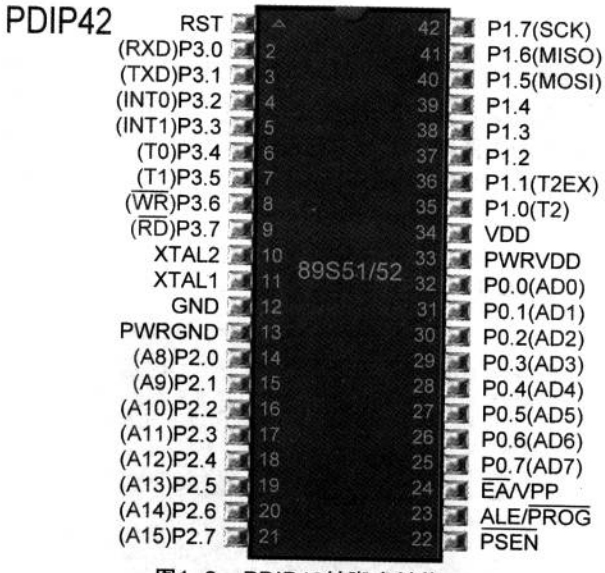


图1-8 PDIP42针脚式封装

双列直插式封装的PDIP40

89C51/89S51的第二种针脚式封装为40个引脚双并排的PDIP40，这种封装与MCS-51完全兼容。PDIP40与PDIP42除引脚数量不同外，尺寸差异也很大，PDIP40刚好可插在面包板或40pin的底座上，如图1-9所示，左上方有个记号的脚为第1脚，然后逆时针排序，分别为第2、3…40脚。相邻两个脚的间距为2.540mm，器件长度为52.578mm，而两排引脚的间距为15.875mm，器件厚度为4.826mm（不含引脚），特别适合学校和培训机构使用。不过，由于针脚式封装体积较大、电路板制作成本较高，目前已经很少使用。

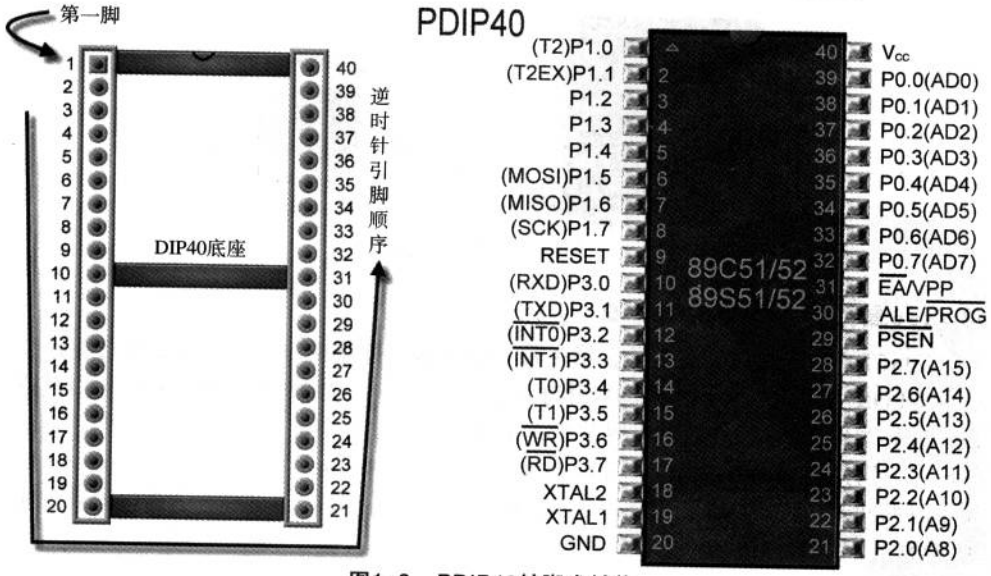


图1-9 PDIP40针脚式封装

除了PDIP42封装，89S51与89C51完全兼容，本书将以PDIP40封装的89S51为探讨对象。当然，要学习8051，笔者强烈建议先将其引脚“背”下来，而要背40个引脚又有点困难，所以在此提供专门的技巧，让大家轻松记住这40个引脚，说明如下。

电源引脚

几乎所有IC都需要连接电源，而8051的电源引脚与大部分数字IC的电源引脚类似，右上角接V_{cc}、左下角接**GND**。所以8051的40脚为V_{cc}引脚，连接5V ± 10%电源，20脚为GND引脚，必须接地。

输入/输出端口

有了电源之后，再来看看8051的主角，也就是输入/输出端口。它们位于刚才所介绍的V_{cc}引脚下面。第39脚为Port 0的第一个引脚，即39脚到32脚等8个引脚为Port 0；Port 0的对面就是Port 1，也就是第1脚到第8脚。Port 1从第1脚开始，所以Port 2从其斜对角第21脚开始，也就是在右下方，21脚到28脚就是Port 2。同样地，Port 2的斜对面就是Prot 3，第10脚到第17脚就是Port 3。**39**、**1**、**21**、**10**就是这4个**Port**的第一个引脚，我们可通过图1-10辅助记忆这4个输入/输出端口。

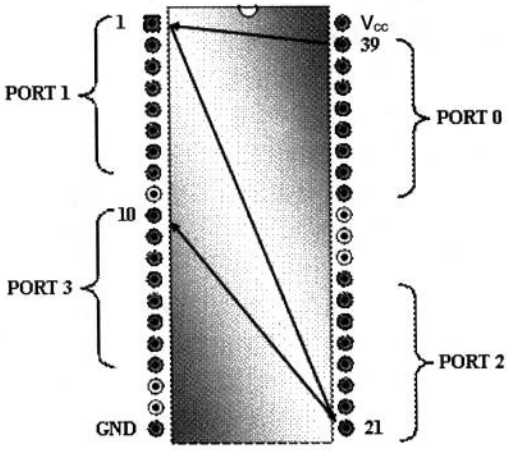


图1-10 引脚辅助记忆图

复位引脚

几乎所有微控制器都需要复位（Reset）的操作，对于8051而言，只要复位引脚接高电平超过两个机器周期（约2μs），即可产生复位的操作。而8051的复位引脚在Port 1与Port 3之间，即第9脚，辅助记忆的方法是“系统久久不动，就要按一下Reset钮复位系统”，这久久就是第9脚的谐音。

频率引脚

微控制器都需要时钟脉冲，而在接地引脚的上方两个引脚，即第19、18脚，就是时

钟脉冲引脚，分别是**XTAL1**、**XTAL2**。

存储器引脚

8051内部有存储器，外部也可接存储器。使用内部存储器还是外部存储器，则需要视第31脚（Port 0下面那个脚）而定！第31脚就是 $\overline{EA}$ 引脚，即访问外部存储器启用（External Access Enable）引脚。当 $\overline{EA}$=1时，系统使用内部存储器；当 $\overline{EA}$=0时，系统使用外部存储器。对于初学者而言，上课时间较少，又有升学的压力，所写的程序应该不会太复杂，大多只使用内部存储器，所以就把第31脚直接接到V_{cc}。若使用无内部存储器的8031/8032（稍后在1.2.4节再详细介绍），则把第31脚接到GND。

外部存储器控制引脚

现在只剩下 $\overline{EA}$ 引脚下面的两个引脚了。这两个引脚与 $\overline{EA}$ 引脚有点类似，都是针对存储器的控制的，说明如下。

- 第30脚为地址锁存启用ALE信号（Address Latch Enable），其功能是在访问外部存储器时，送出一个将原本在Port 0的地址（A0~A7）地址锁在外部锁存器IC（如74373）的信号，让Port 0空出来，以传输数据。简单地讲，当外接存储器电路时，若ALE=1，P0被当成地址总线；若ALE=0，P0被当成数据总线。
- 第29脚为程序储存启用信号 $\overline{PSEN}$（Program Store Enable），其功能也是访问外部存储器。通常把此引脚连接到外部存储器（ROM）的 $\overline{OE}$ 引脚，当8051要读取外部存储器的数据时，此引脚就会输出一个低电平信号。

相对于前面的38个引脚，第29、30脚比较难以说明。所幸，只要不动用外部存储器，就可当它们不存在！在后面采用外部存储器的章节里再进行说明。

根据上述要诀，连续三天，睡觉前回忆一下、起床后再想一想；则，想忘掉，难！不想记住，更难！

另外，或许有人会质疑，“有这么简单吗”，当然没这么容易！8051的40个引脚里，其中有很多是复用引脚，简单讲就是多用途的引脚。以第32~39脚为例，平时为Port 0；若是连接外部存储器时，则当成AD0~AD7引脚，而AD0~AD7就是地址引脚与数据引脚混合的复用引脚，好像有点复杂，但如果不接外部存储器，就可以忽略它。

1.2.3 8051的基本电路

所谓“基本电路”，是指8051电路工作不可或缺的基本连接线路。所以，这是必要的！当然，在此我们也有熟记基本电路的方法。基本电路包括4部分，详见如下说明。

先接电源

没有电路不需要电源的，8051电路也是如此！首先将第40脚接V_{cc}，也就是+5V电源，第20脚接地GND。如图1-11所示。

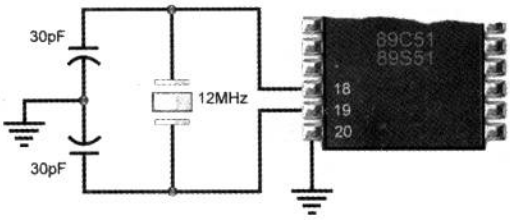


图1-11 使用内部振荡电路

再接时钟脉冲

8051内部已具备振荡电路，只要在GND引脚上方的两个引脚（即第19、18脚）连接简单的石英振荡晶体（Crystal）即可。至于8051的时钟脉冲频率，目前的MCS-51芯片的工作频率已大幅提高，例如Atmel公司的89C51的工作频率为0~24MHz，而华邦电子（**Winbond**）更提供了40MHz的版本，未来必然还会有更高频率的版本！尽管如此，目前还是采用12MHz时钟脉冲。如果懒得再设计一个振荡电路，则按图1-11连接即可。如果非得自行设计一个振荡电路不可，则可按图1-12连接。

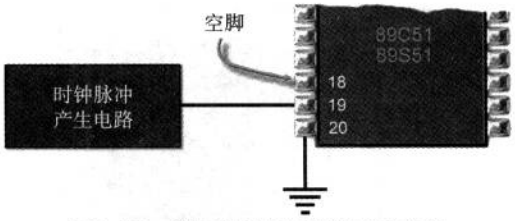


图1-12 使用外部时钟脉冲产生电路

复位电路

8051的复位引脚（Reset）是第9脚，当此引脚连接高电平超过2个机器周期（1个机器周期包含12个时钟脉冲），即可产生复位的操作。以12MHz的时钟脉冲为例，每个时钟脉冲为1/12μs，2个机器周期为2μs。因此，我们可在第9脚上连接一个可让该引脚上产生一个2μs以上的高电平脉冲即可产生复位的操作，如图1-13的左图所示。

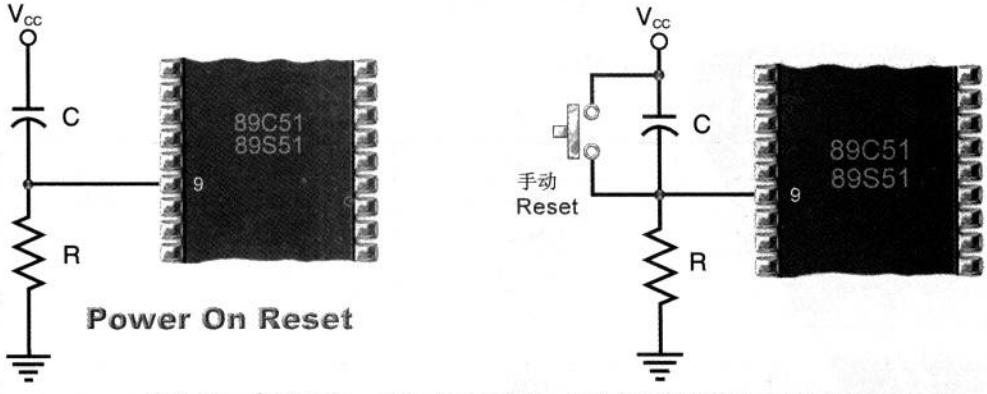


图1-13 左边为Power On Reset电路，右边为含手动的Reset电路

接上电源的瞬间，电容C上没有电荷，相当于短路，所以第9脚直接连接到V_{CC}，即8051执行复位操作。随着时间的增加，电容上的电压逐渐增加，而第9脚上的电压逐渐下降，当第9脚上的电压降至低电平时，8051即恢复正常状态，称之为“**Power On Reset**”。在此使用10kΩ电阻、10μF电容，其时间常数远大于1μs，所以第9脚上的电压可保持2μs以上的高电平，足以使系统复位。当然，只要时间常数大于2μs即可，而不一定要使用10kΩ电阻和10μF电容。

通常，我们还会在电容两端并接一个按钮开关，如图1-13的右图所示，此按钮开关就是一个手动的Reset开关（强制Reset）。

存储器设置电路

基本电路的最后一个部分是存储器的设置，如果把第31脚（$\overline{EA}$）接地，则采用外部存储器；如果把第31脚（$\overline{EA}$）接Vcc，则采用内部存储器。在本书里大多采用内部存储器，所以把第31脚与第40脚及Vcc相连接。整个基本电路如图1-14所示。

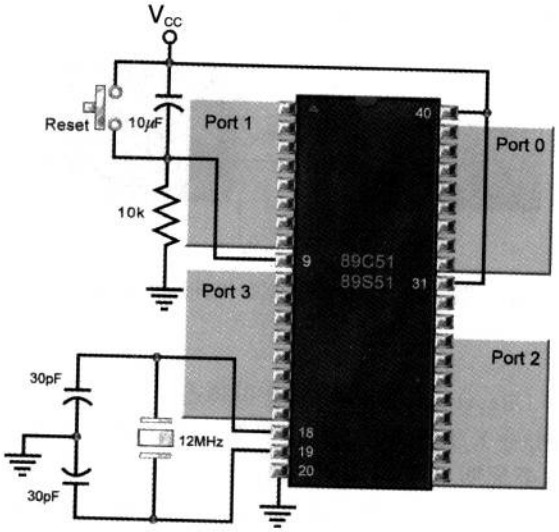


图1-14　基本电路

图1-14中所采用的器件如表1-1所示。

表1-1

项次	名　称	规　格	数　量	备　注
1	89C51	12MHz	1个	或89S51
2	石英振荡晶体	12MHz	1个	
3	电解电容	10μF/25V	1个	
4	陶瓷电容	30pF	2个	
5	电阻	10kΩ	1个	
6	按钮开关	a接点	1个	TACK SW.

1.2.4 MCS-51系列

一般来说，**MCS-51**系列（Micro Controller System，简称**MCS**）可分为51与52两大类。52类可说是51类的增强型，其最大的特色就是内部存储器容量更大，增加了一个定时器/计数器。

若按芯片内的ROM来区分，**MCS-51**系列可分为无ROM型（8031/8032）、Mask ROM型（8051/8052）、EPROM型（8751/8752）及EEPROM型（89C51/89C52、89S51/89S52），如表1-2所示。

表1-2

	51类				52类			
型号	**8031**	**8051**	**8751**	**89C51 89S51**	**8032**	**8052**	**8752**	**89C52 89S52**
类型	无ROM	Mask ROM	EP ROM	EEP ROM	无ROM	Mask ROM	EP ROM	EEP ROM
ROM	内部0KB 外接 64KB	内部4KB 外接最大64KB			内部0KB 外接 64KB	内部8KB 外接最大64KB		
RAM	内部128B 外接最大64KB				内部256B 外接最大64KB			
定时器/ 计数器	2个16 位定时器/计数器				3个16 位定时器/计数器			
中断源	5				6			
I/O	4个8位输入/输出端口				4个8位输入/输出端口			

以下简述这几种8051。

无ROM型

8031/8032为无ROM型单片机，使用这种单片机必须外接程序存储器。由于其封装成本与含ROM型的单片机很接近，且需要外接程序存储器，反而使电路成本大增。目前除了程序太大，无法完全放入单片机外，已很少有人采用此类型的单片机。如图1-15所示。

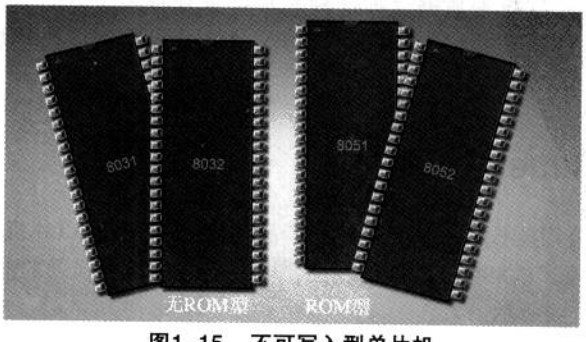


图1-15 不可写入型单片机

Mask ROM型

8051/8052为Mask ROM型单片机，这种单片机直接将程序放入芯片中的程序存储器，所以不必要“烧录”程序（也不能烧录），而且单价低廉。但由于这种芯片必需要制作其独有的光罩（Mask），必须要批量生产。例如键盘里所用的单片机（8048，是8051的前一代）就是Mask ROM型单片机。

EPROM型

8751/8752为EPROM型单片机，这种单片机可将程序烧入芯片中的程序存储器，也可用紫外线清除程序存储器里的数据，所以可重复使用不同的程序。IC上面有一个窗口，可看到内部的芯片与连接线，通常在烧录完毕后，在窗口上贴黑色胶布，以防止数据消失。如要清除ROM里的数据，则使用紫外线照射窗口15~30分钟即可。由于这种封装成本较高，再加上其清除操作麻烦且费时，目前几乎不再生产这种器件了，如图1-16所示。

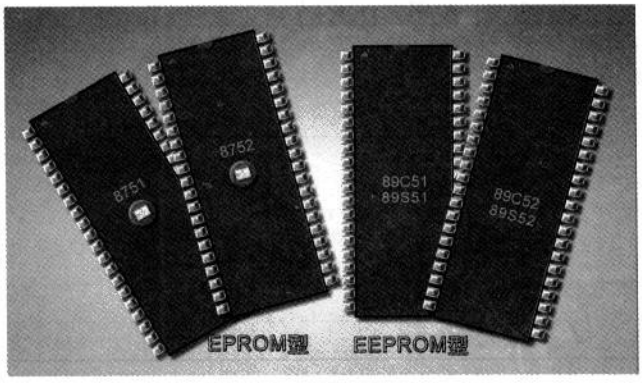


图1-16 可重复写入型

EEPROM型

89C51/89C52、89S51/89S52为EEPROM型单片机，这种单片机都可以将程序“加载”到芯片中的程序存储器。所不同的是，**89C51/89C52**是用**5V**及**12V**电压烧录与清除程序存储器里的数据，而**89S51/89S52**只要**5V**电压即可烧录与清除，因此早已成为主流器件。厂商的技术数据宣称这种芯片可重复写入与清除上千次。而依笔者的经验，除非操作上的错误或将引脚折断，否则就算是很用力地给它烧，都还很难给它烧掉！

1.2.5 关于Atmel的51系列

Atmel半导体公司所提供的89C51系列产品是MCS-51兼容单片机中的主流产品。在89C51系列中，已停产的89C51最具代表性，取而代之的是更强劲的89S51，这两种单片机的结构比较如表1-3所示。

表1-3

型 号	89C51/52	89S51/52
位数	8位	8位
工作频率	0~24MHz	**0~33MHz**
ROM	4KB/8KB	4KB/8KB
RAM	128 B/256 B	128 B/256 B
I/O	4个8位输入/输出端口	4个8位输入/输出端口
定时器/计数器	2/3个16位定时器/计数器	2/3个16位定时器/计数器
Watchdog Timer		**14**位看门狗计数器
中断源	5/6个	**6/8**个
串行口	一组全双工万用串行口UART	一组全双工万用串行口UART
省电模式	Idle模式及Power-down模式	Idle模式及Power-down模式
数据指针寄存器	一组16位数据指针寄存器	两组**16**位数据指针寄存器

由上表可得知，只要89C51/52有的，89S51/52都有！而其中比较特别的是89S51/52新增一组14位看门狗计数器（**Watchdog Timer**，简称WDT）。虽然89S51/52的工作频率提高为0~33MHz，但实质性能提升不大！就像0~24MHz的89C51/52一样，还是会应用12MHz的工作频率，这样才能直接沿用原有的程序设计；另外，设计程序时也比较容易计算，且耗用的资源也比较少。

虽然89C51停产了，但其核心仍存在于89S51及许多89C51的增强版本。

- AT89C51RC单片机具有32KB程序存储器、512 B数据存储器、Watchdog Timer…，除存储器比较多外，都与89S51相同。
- AT89C51CC001、AT89C51CC002、AT89C51CC003等单片机以89C51为核心，并扩展外围设备，除配置更多的存储器外，还增加了10位的ADC[1]、CAN[2]控制器等，而其重复数据烧录/清除的次数更可达到10万次。

[1] ADC为Analog to Digital Converter的简称，也就是将模拟信号转换成数字信号的转换器。

[2] CAN为Controller Area Network的简称，这是一种微控制器与CAN总线的接口，而CAN控制器应用BOSCH CAN 2.0B Data Link Layer Protocal通信协议。

1.3 了解MCS-51的存储器结构



除了无ROM型的8031和8032外，MCS-51的存储器包括程序存储器（ROM）与数据存储器（RAM）两部分，这两部分一般是独立的。标准的8x51系列具有4KB程序存储器、128 B数据存储器，而标准的8x52系列具有8KB程序存储器、256 B数据存储器，刚好是8x51系列的两倍。不管是8x51、8031、8032或8x52，其外部扩展的程序存储器或数据存储器最多为64KB。

MCS-51的兼容单片机都增大了其内部程序存储器与数据存储器。例如Atmel半导体公司的TS83C51RB2，其内部有16KB程序存储器、256 B数据存储器；TS83C51RC2，其内部有32KB程序存储器、256 B数据存储器；TS83C51RD2，其内部有64KB程序存储器、768 B数据存储器。尽管如此，在此仍探讨基本的MCS-51单片机的标准存储器结构。

1.3.1 程序存储器

顾名思义，程序存储器（ROM）是存放程序的，CPU将自动从程序存储器读取所要执行的指令码。MCS-51可选择使用内部程序存储器或外部程序存储器，具体说明如下。

- 若使用8031或8032，由于内部没有程序存储器，一定要使用外部程序存储器，所以其 $\overline{EA}$ 引脚必须接地。
- 当 $\overline{EA}$ 引脚接高电平时，CPU将使用内部程序存储器，若程序超过4KB（8x51）或8KB（8x52），则CPU会自动从外部程序存储器里读取超过部分的程序代码。
- 当 $\overline{EA}$ 引脚接地时，CPU将自外部程序存储器读取所要执行的指令码，而CPU内部的程序存储器形同虚设。

> 1. 对于初学者而言，4KB的程序存储器已是绰绰有余。
> 2. 坏掉的8x51/8x52，很可能是其中的程序存储器坏掉了，可将其 $\overline{EA}$ 引脚接地，改外接程序存储器，即可当作8031/8032使用。

MSC-51的程序存储器结构如图1-17所示。

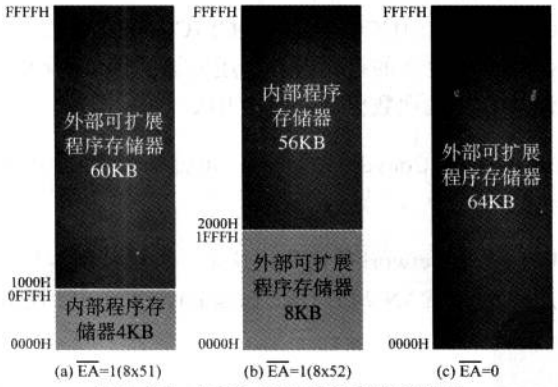


图1-17 MCS-51的程序存储器结构

当CPU复位后，程序将从程序存储器0000H位置开始执行，如没有遇到跳跃指令，则沿程序存储器顺序执行。当然，程序存储器前面几个位置还有一些玄机，留待有中断的章节再详细说明。

1.3.2 数据存储器

MCS-51的程序存储器与数据存储器是分开的独立区域，所以访问数据存储器时，所

使用的地址并不会与程序存储器发生冲突。相对于程序存储器，数据存储器就比较不简单，其示意图如图1-18所示。

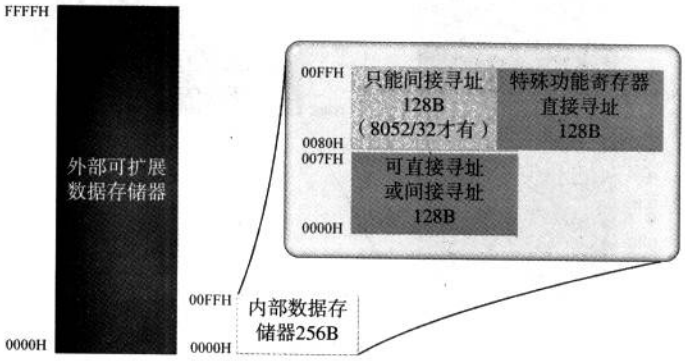


图1-18 MCS-51的数据存储器结构

8051里的数据存储器除内部数据存储器外，还可扩展外部数据存储器，这两部分的数据存储器可以并存。不过，访问数据存储器时，所采用的指令并不一样。例如访问内部数据存储器时，可用MOV指令，但访问外部数据存储器时，则使用MOVX指令。

另外，内部数据存储器也有点文章！说明如下。

▶ 从0000H到007FH之间的128B为可直接寻址或间接寻址的存储器。而“直接寻址”与“间接寻址”在编写C语言程序时可以用数据类型来区分。在这一区里的数据存储器又可分成3部分，说明如下。

• 寄存器组区

0000H到001FH的32个地址为寄存器组（Register Bank）区，具体说明如下。

1. 0000H到0007H为寄存器组0（即**RB0**），0008H到000FH为寄存器组1（即**RB1**），0010H到0017H为寄存器组2（即**RB2**），0018H到001FH为寄存器组3（即**RB3**）。

2. 每个寄存器组都包含R0、R1…R78个寄存器，而任何一个时间只能使用其中一个寄存器组。

3. 寄存器组的切换可由程序状态字（**P**rogram **S**tatus **W**ord，简称**PSW**）中的**RS1**与**RS0**来决定，如表1-4所示。

表1-4

RS1	RS0	寄存器组	位 址
0	0	RB0	0000H~0007H
0	1	RB1	0008H~000FH
1	0	RB2	0010H~0017H
1	1	RB3	0018H~001FH

4. 当CPU复位时，系统的堆栈指针（SP）指向07H地址。所以数据存入堆栈时，将从08H开始，也就是RB1里的R0地址。为避免冲突或不必要的错误，通常会把堆栈指针移到30H以后的地址。

8051内部数据存储器结构如图1-19所示。

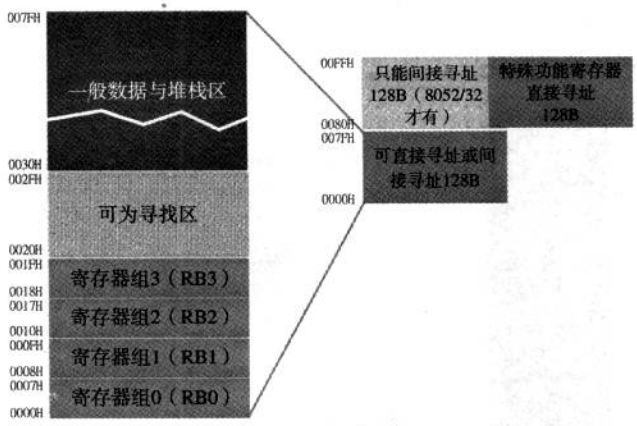


图1-19　内部数据存储器

• 可位寻址区

0020H到002FH的16个字节存储器区为可位寻址区。通常访问存储器是以字节为单位，“可位寻址”是指可以指定访问某个位（bit）。在8051的汇编语言里，可以使用布尔运算指令进行位操作，例如要把20H存储器地址的bit 5设为1，则可以使用下列指令。

```
SETB 20H.5
```

另外，从0020H到002FH的16个字节，总共128个位（16×8），也可以直接指定为0到127，对刚才的20H存储器地址的bit 5而言，也可以将“20H.5”指定为“05”，则可以使用下列指令。

```
SETB 05
```

同理，若要将25H存储器地址的bit 2清0，则可以使用下列指令。

```
CLR 25H.2
```

或（5×8+2=42）

```
CLR 42
```

• 一般数据与堆栈区

0030H到007FH的80个字节地址为一般数据访问及堆栈区。由于CPU复位后，堆栈指针指向07H位置，为了确保数据的安全与程序执行的正确，如果在程序之中使用了PUSH、POP命令，最好能把堆栈指针改至本区。例如要将堆栈指针移至0030H地址，则在程序开始处应使用如下指令。

```
MOV SP, #30H
```

▶ 从0080H到00FFH之间的128 B为特殊功能寄存器（Special Function Register，简称SFR），或可直接寻址的存储器，至于“特殊功能寄存器”，稍后再详细说明。

▶ 如果是8052/8032，则0080H到00FFH之间的128B，除了特殊功能寄存器或可直接寻址的存储器外，也可以用间接寻址的方式，访问与这特殊功能寄存器位置重叠，但是可以独立访问的存储器。

1.3.3 特殊功能寄存器

	8	9	A	B	C	D	E	F	
F8									FF
F0	B								F7
E8									EF
E0	ACC								E7
D8									DF
D0	PSW								D7
C8	T2CON		RCAP2L	RCAP2H	TL2	TH2			CF
C0									C7
B8	IP								BF
B0	P3								B7
A8	IE								AF
A0	P2		AUXR1				WDTRST		A7
98	SCON	SBUF							9F
90	P1								97
88	TCON	TMOD	TL0	TL1	TH0	TH1	AUXR		8F
80	P0	SP	DP0L	DP0H	DP1L	DP1H		PCON	87
	0	1	2	3	4	5	6	7	

[1] 其中C8一行为8052/8032才有的寄存器，其中左数第一列（对应列号为8）为可位寻址的寄存器，灰底的部分为89S51/52才有的。

[2] 8051/52、89C51/52只有一组数据指针寄存器，所以其中的DP0L应改为DPL、DP0H应改为DPH。

在MCS-51里，寄存器只是CPU里特定地址的数据存储器而已。而在0080H到00FFH之间的128 B，正是特殊功能寄存器（Special Function Register，简称**SFR**）所在位置。而什么是“特殊功能寄存器”呢？特殊功能寄存器就是8051内部的装置，若用汇编语言编写程序，我们必须确切地掌控这些寄存器。若以C语言编写程序，就不是那么重要，其位置的声明放在Keil C所提供的“reg51.h”头文件里，我们只要把它包含到程序里即可，而不必记忆这些位置。以下简单介绍这些寄存器，仅供参考，若教学时间不够，可直接跳过。

P0、P1、P2、P3

P0~P3为MCS-51的4个输入/输出端口，其地址分别为80H、90H、0A0H及0B0H，在

第3章再详细介绍这4个输入/输出端口。

SP

SP为堆栈指针寄存器（**S**tack **P**ointer register），其地址为81H。堆栈是一种特殊的数据储存方式，其数据的操作顺序是先进后出（**F**irst **I**n **L**ast **O**ut，简称为**FILO**）。当数据以PUSH命令送入堆栈时，SP自动加1；若以POP命令从堆栈取出数据时，SP自动减1。当然，使用C语言编写程序时，几乎可以不管这个寄存器。

DPL、DPH

89C51只有一组16位的数据指针寄存器（**D**ata **P**ointer register，简称**DPTR**），而这组数据指针寄存器是由DPL与DPH两个8位的数据指针寄存器组成，其地址分别为82H、83H。若以DPL为低8位、DPH为高8位，所组成的16位数据指针寄存器，将可寻址到64KB的数据地址。89S51有两组16位数据指针寄存器，分别是DP0L、DP0H、DP1L及DP1H，其地址分别为82H、83H、84H、85H。若用汇编语言编写程序，DPTR是查表法必须用到的寄存器！不过，使用C语言编写程序时，就不太需要由我们直接控制这个寄存器。

PCON

PCON为电源控制寄存器（**P**ower **C**ontrol register），其地址为87H，其功能是设置CPU的电源模式，7.3.3小节将再对其进行说明。

TCON

TCON为定时器/计数器控制寄存器（**T**imer/**C**ounter **Con**trol register），其地址为88H，其功能是设置定时器/计数器的启动、记录定时器/计数溢出位以及外部中断的触发方式等，后续关于定时器/计数器部分将再详细说明。

TMOD

TMOD为定时器/计数方式控制寄存器（**T**imer/**C**ounter **Mode** **C**ontrol register），其地址为089H，其功能是设置定时器/计数的方式，在定时器/计数器部分将进行详细说明。

TL0、TL1、TH0、TH1

TL0、TH0为第一组定时器/计数器（Timer0）的计量寄存器，其地址为8AH、8CH，将TH0与TL0组合即可进行16位的定时/计数。TL1、TH1为第二组定时器/计数器（Timer1）的计量寄存器，其地址为8BH、8DH，将TH1与TL1组合即可进行16位的定时/计数，在定时器/计数器部分将进行详细说明。

SCON

SCON为串行端口控制寄存器（**S**erial port **Con**trol register），其地址为98H，其功能

是设置串行端口工作方式与标志，在串行口部分将进行详细说明。

SBUF

SBUF为串行端口寄存器（Serial BUFfer），其地址为99H，它由使用同一个地址的两个寄存器构成，其中一个寄存器作为传送数据用的寄存器，另一个寄存器作为接收数据用的寄存器。至于如何分辨同一个地址的两个寄存器，则视指令而定，若是数据传出的指令，则自动定位到传出数据用的寄存器；若是接收数据的指令，则自动定位到接收数据用的寄存器，在串行口部分将进行详细说明。

IE

IE为中断启用寄存器（Interrupt Enable register），其地址为0A8H，其功能是启用中断功能，在中断部分将进行详细说明。

IP

IP为中断优先等级寄存器（Interrupt Priority register），其地址为0B8H，其功能是设置中断的优先等级，在中断部分将进行详细说明。

T2CON

T2CON为Timer 2的定时器/计数器控制寄存器，其地址为0C8H，其功能是设置Timer 2的启动、记录定时器/计数溢出以及外部中断的触发方式等。只有在8052/8032才有Timer 2。

RCAP2L、RCAP2H

RCAP2L、RCAP2H为捕捉寄存器（Capture register），其地址为0CAH、0CBH。当Timer 2在捕捉方式时，若T2EX（P1.1）引脚上的输入信号由高电平转为低电平，TL2与TH2的内容将被载入RCAP2L与RCAP2H里，就像是把Timer 2的内容捉进RCAP寄存器一样。只有8052/8032才有Timer 2。

TL2、TH2

TL2、TH2为第三组定时器/计数器（Timer2）的计值寄存器，其地址为0CCH、0CDH，将TH2与TL2组合即可进行16位的定时器/计数。只有8052/8032才有Timer 2。

PSW

PSW为CPU的程序状态字寄存器（Program Status Word register），其地址为0D0H，其内容说明如下。

	7	6	5	4	3	2	1	0
PSW	CY	AC	F0	RS1	RS0	OV		P

- **PSW.7**：本位为进位标志位（**CY**），进行加法（减法）运算时，若最左边位（MSB，即bit 7）产生进位（借位）时，则本位将自动设为1，即CY=1；否则CY=0。

- **PSW.6**：本位为辅助进位标志位（**AC**），进行加法（减法）运算时，若bit 3产生进位（借位）时，则本位将自动设为1，即AC=1；否则AC=0。
- **PSW.5**：本位为用户标志位（**F0**），可由用户自行设定。
- **PSW.4**与**PSW.3**：这两个位为寄存器组选择位（**RS1**、**RS0**），其功能如下表所示：

PSW.4(RS1)	PSW.3(RS0)	寄存器组
0	0	RB0
0	1	RB1
1	0	RB2
1	1	RB3

- **PSW.2**：本位为溢出标志位（**OV**），当进行算术运算时，若发生溢出，则OV=1；否则OV=0。
- **PSW.1**：本位为保留位，没有提供服务。
- **PSW.0**：本位为奇偶标志位（**P**），8051采用偶同位，若ACC里有奇数个1，则P=1；若ACC里有偶数个1，则P=0。

ACC

ACC累加器（Accumulator）又称为A寄存器，其地址为0E0H，这个寄存器提供CPU主要操作的位置，可以说是最常用的寄存器。

B

B寄存器的地址为0F0H，主要功能是搭配A寄存器进行乘法或除法运算，进行乘法运算时，乘数放在B寄存器，而运算结果的高八位放在B寄存器；进行除法运算时，除数放在B寄存器，而运算结果的余数放在B寄存器。若不进行乘/除法运算，B寄存器也可当成一般寄存器使用。

AUXR

AUXR寄存器为89S51新增的辅助寄存器（**AUX**iliary **R**egister），其地址为8EH，其内容说明如下。

	7	6	5	4	3	2	1	0
AUXR				WDIDLE	DISRTO			DISALE

- **WDIDLE**：本位设置在闲置模式（Idle Mode）下是否启用“看门狗”。若本位设为1，则在Idle模式下将启用“看门狗”。若本位设为0，则在Idle模式下将不使用“看门狗”。
- **DISRTO**：本位设置是否输出复位信号。若本位设为1，则Reset引脚（第9脚）只有输入功能。若本位设为0，则在WDT计数完毕后，Reset引脚输出复位信号（即高电平脉冲）。
- **DISALE**：本位设置是否启用ALE信号。若本位设为1，则只有在执行MOVX指令

或MOVC指令时，ALE引脚（第30脚）才会正常工作。若本位设为0，则固定每6个脉冲就输出1个高电平脉冲，稍后对其进行说明。

其他位为保留位。当然，这个寄存器只有在89S51里才有。

AUXR1

AUXR1寄存器为89S51新增的第二个辅助寄存器，其地址为0A2H，其内容说明如下。

	7	6	5	4	3	2	1	0
AUXR1								DPS

- **DPS**：本位的功能是选择数据指针寄存器。若本位设为1，则使用DP1L及DP1H。若本位设为0，则使用DP0L及DP0H。

其他位为保留位。同样地，这个寄存器只有在89S51里才有作用。

WDTRST

WDTRST寄存器为89S51新增的看门狗定时器复位寄存器（Watchdog Timer Reset register），其地址为0A6H。当我们要启用看门狗定时器WDT时，按顺序将01EH、0EH放入WDTRST寄存器，当14位计数器溢出（达到16383，即3FFFH），即由RESET引脚送出一个高电平脉冲使装置复位。此脉冲的宽度为98×TOSC，其中TOSC=1/FOSC，以12MHz的时钟脉冲为例，脉冲的宽度为 $\text{Width} = 98 \times \frac{1}{12\text{M}} \cong 8.167\mu\text{s}$，关于“看门狗”与省电模式，在后续相关章节将进行说明。

1.4 8051的时序分析与复位



在本单元里将介绍8051的复位（RESET）与时序分析。

1.4.1 时序分析

时钟脉冲是微型计算机的基本信号，在1.2节里我们曾经简单地介绍了8051的时钟脉冲。不管是采用内部的振荡电路，还是由外部的时钟脉冲产生电路提供时钟脉冲，这个时钟脉冲将成为整个系统工作的基础。89C51的额定时钟脉冲为0~24MHz，表示只要不超过24MHz都是可以的！而89S51的额定时钟脉冲为0~33MHz，表示只要不超过33MHz就不会有问题。当我们在设计电路时，是不是要使用其最高的频率呢？这不是一般的个人计算机，不必要太快（超频）！若时钟脉冲的频率太高，可能会导致程序太复杂，使用的CPU资源太多，“延迟函数”就是最明显的例子！

通常我们会挑选一个常用、容易买到（且便宜）的石英振荡晶体，而且程序不必刻意修改就能兼容。既然如此，那就挑选最常用的12MHz时钟脉冲吧！

图1-20所示为12MHz时钟脉冲的时序图。一个机器周期（machine cycle）是由6个状态周期（S1到S6）所构成的，每个状态周期包括两个时钟脉冲（即P1、P2）。对于12MHz的时钟脉冲而言，一个脉冲的周期为1/12μs，一个机器周期包含12个时钟脉冲，也就是1μs。

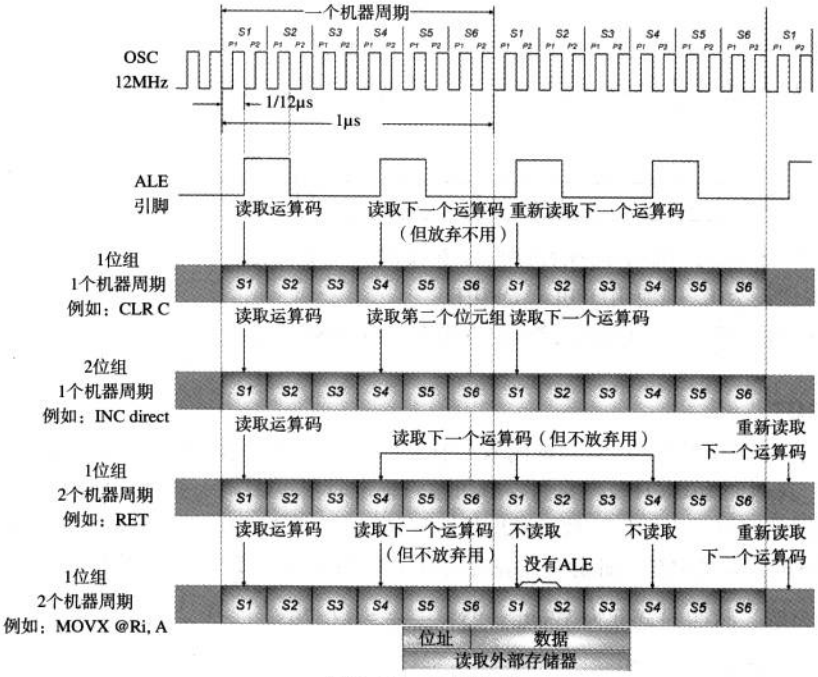


图1-20　时序分析

在8051的111个指令里，除了执行乘法与除法指令需要4个机器周期外，其余指令都能在1个或2个机器周期执行完毕。尽管如此，有些指令的长度为1 B、有些为2 B，还有少数指令为3 B。对于不同的指令，CPU如何读取（fatch）与执行（execution）呢？在此将配合在图1-20简要说明。首先是地址锁存启用引脚ALE，每个机器周期送出两个脉冲（分别是在S1及S4时），以锁存P0输出的地址（A0~A7），CPU将进行读取存储器的操作。而不同的指令类型，其操作分别说明如下。

1. 1个机器周期、1B的指令，如CLRC指令，在S1时读取指令、S6执行完毕；而在S4时读取下个指令，但并不使用它，直到下个机器周期的S1时，再重新读取下个指令。

2. 1个机器周期、2B的指令，如INC direct指令，在S1时读取指令、S4时读取第二个指令、S6执行完毕。在下个机器周期的S1时读取下个指令……，依次类推。

3. 2个机器周期、1B的指令，如RET指令，在S1时读取指令，而在S4、下个机器周期的S1、S4时分别读取下个指令，由于指令尚未执行完毕，所以这三个阶段的指令读取，都会被放弃。直到第二个机器周期的S6，指令执行完毕后，CPU才会在第三个机器周期的S1，重新读取下个指令，这才是有效的读取。

4. 另外一种2个机器周期、1B的指令为存取外部存储器数据的指令，即MOVX指令。同样在第一个机器周期的S1时读取指令，而在S4读取的下个指令，当然也会被放弃。S5时P0送出的A0到A7地址将被放入锁存器，而S6到下个机器周期的S3之间，即由P0进行外

部存储器的数据存取。由于进行外部存储器的存取，第二个机器周期的S1与S4不进行读取指令的操作。直到第三个机器周期的S1，才会重新读取下个指令。

1.4.2 复位

对于微型计算机而言，复位RESET是一项很重要的归零调整操作。8051的RESET是将高电平加到RESET引脚（第9脚）上，并保持时间超过两个机器周期以上，也就是2μs。用手按8051系统里的RESET按钮开关，不管手有多快都会超过2μs，换言之，只要按RESET按钮，就一定会使系统复位！当系统复位时，CPU内部寄存器将回归初始状态（状态如表1-5所示），程序将从0000H处开始执行。

表1-5

寄存器	状 态	寄存器	状 态
ACC	00000000B	TMOD	00000000B
B	00000000B	TCON	00000000B
PSW	00000000B	T2CON	00000000B
SP	00000111B	TH0	00000000B
DPTR： DPH DPL	 00000000B 00000000B	TL0	00000000B
		TH1	00000000B
		TL1	00000000B
P0	11111111B	TH2	00000000B
P1	11111111B	TL2	00000000B
P2	11111111B	RCAP2H	00000000B
P3	11111111B	RCAP2L	00000000B
IP： 8051 8052	 XXX00000B XX000000B	SCON	00000000B
		SBUF	未定
IE： 8051 8052	 0XX00000B 0X000000B	PCON： NMOS CHMOS	 0XXXXXXXB 0XXX0000B
		PC	0000H

1.5 8051的开发流程与工具



8051系统的开发流程与一般单片机的开发流程类似，其基本开发流程可分为软件与硬件两部分，而这两部分是并行开发的。在硬件开发方面，主要是设计电路板（prototype），也就是目标板（target board）。在软件开发方面，则是编写源程序（可使用C语言或汇编语言）、编译、组译生成可执行文件，然后进行调试/仿真。当完成软件设计后，即可应用在线仿真器（**In-Circuit Emulator**，简称**ICE**）上加载该可执行程序，然后

在目标板上进行在线仿真。若软、硬件设计无误，则可利用IC烧录器，将其可执行码烧录到89S51，最后将该89S51插入目标板，即完成设计，如图1-21所示。

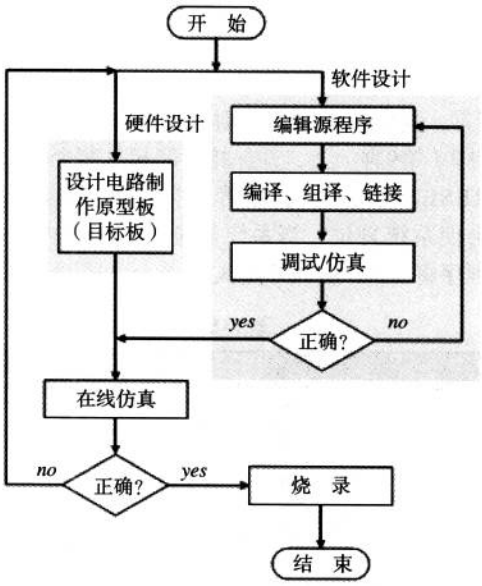


图1-21 8051系统的开发流程

8051的开发工具非常多，在此只介绍3种较普及的工具。

1.5.1 传统开发工具

当我们要应用8051来设计控制电路时，除了8051的电路设计外，还得编写8051程序。传统的8051程序开发大多是在DOS环境下完成，而在Windows环境中开发8051程序反倒有点不太方便！

如图1-22所示，其中各步骤说明如下。

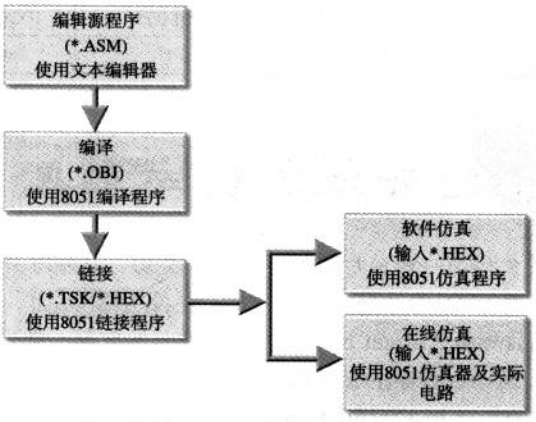


图1-22 传统的8051程序开发流程

编辑源程序

首先利用文本编辑器来编写源程序（source code，即*.asm），这是用8051汇编语言所编写的程序，是人们能理解的文本文件。若是在DOS环境下编写源程序，可使用PE2之类的文本编辑器；若是在Windows环境下编写源程序，则可使用Windows内置的记事本程序。

编译与链接

当源程序编辑完成后，紧接着利用8051的编译程序，如美国2500AD公司的x8051，将源程序编译而得到目标文件（object code，即*.obj）；再利用链接程序（Link），将目标文件链接而产生可执行文件（*.TSK）或Intel的十六进制文件（*.HEX）。由于编译与链接是利用两个不同的程序，在DOS下依次进行的操作。我们可使用批处理程序来简化操作程序，如下所示为test.bat批处理程序，其功能是让我们一次操作就可完成编译与链接两个操作。

```
x8051 %1.asm -D
@ECHO OFF
IF ERRORLEVEL 1 GOTO ERR
@ECHO ON
LINK-C%1.obj
@ECHO OFF
GOTO END
:ERR
@ECHO Error……
:END
@ECHO OFF
```

test.bat

若要编译与链接ch1.asm源程序，则在DOS命令行下输入。

test ch1 Enter

注意：请不要指定扩展文件名，这样可以省去不少麻烦。

软件仿真

产生TSK文件或HEX文件后，可利用8051的软件仿真程序（如AVSIM51）进行简单的软件仿真。

在线仿真

在线仿真是将TSK文件或HEX文件加载到在线仿真器，再把在线仿真器当作8051，插入所开发的目标板上，即可进行在线仿真。当然8051在线仿真器是不可缺的！而8051在线仿真器的品牌、种类繁多，而在附录A里介绍的8051在线仿真器，属于较新一代的8051在线仿真器。

如果一切都正确，则可利用**IC**烧录器，将HEX文件烧录到89C51/89S51，该

89C51/89S51就含有我们所编写的程序，将它插在目标板上，就完成此8051电路设计。

1.5.2 Altium Designer电路设计软件

Altium Designer为一套多功能的电路设计软件，其中最著名的有两部分，第一部分是电路图/电路板的设计工具，也就是大家熟悉的**Protel**电路板设计软件。第二部分是FPGA嵌入式系统（embedded system）设计，即**TASKING**。实际上，**TASKING**拥有超过10万个客户，包括分布于全世界知名的通信厂商、资通厂商、无线厂商与外围装置厂商等。我们可以应用这套软件来开发FPGA嵌入式系统、单片机的软件与硬件。**TASKING**除了提供整合式文本编辑器外，还提供编译、调试、嵌入式因特网及**RTOS**（实时操作系统，Real Time Operation System）等，并支持多个DSP、8位、16位及32位的嵌入式微控制器。

在**TASKING**里，嵌入式软件的建构流程与一般微机或单片机系统的软件建构流程类似，基本流程都是“编写源程序”→“编译（C语言）”→“组译”→“链接”→“仿真/调试”→“在线仿真”。如图1-23所示是在**TASKING**里的开发FPGA嵌入式系统设计或8051单片机设计的开发流程。

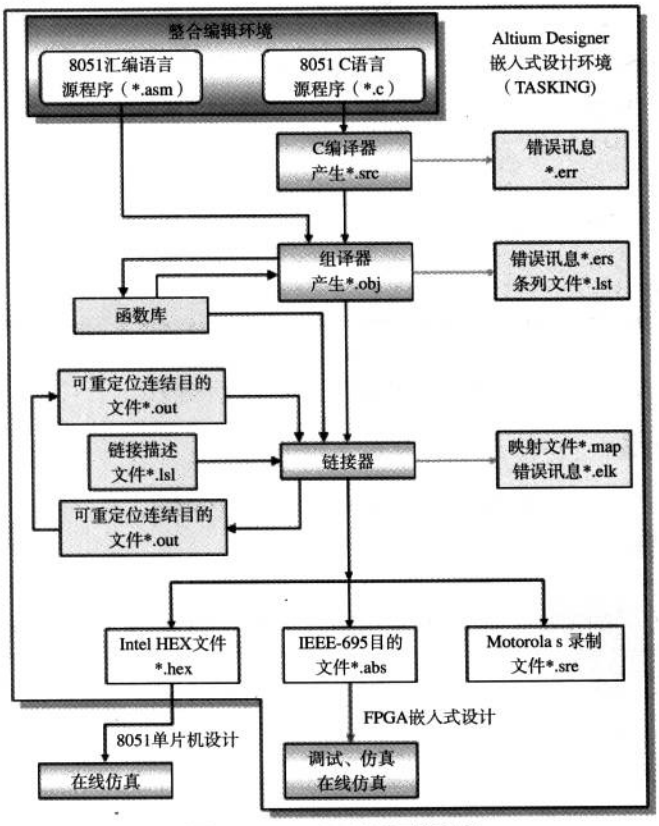


图1-23 TASKING开发流程

如果是使用C语言编写源程序，则需要先经过C语言的编辑器将源程序编译而得到*.src

文件；若在编译的过程中有任何的错误状况，将会把错误讯息存入文件，即*.err文件。*.src文件与汇编语言编写的源程序相同，可经由编译器将它编译而产生目的文件（*.obj）。另外，编译过程也会产生错误信息文件（*.ers）及条列文件（*.lst）。紧接着，再利用链接器链接目的文件与函数库（若在源程序里应用了函数库里的函数），即可产生可执行文件，其中包括3种可执行文件的格式，如Intel的16进制文件（*.hex）、IEEE-695目的文件（*.abs）及Motorola S录制文件（*.sre）。而**TASKING**里的调试/仿真及在线仿真，就是以IEEE-695目的文件为输入。当然，若要使用外部的在线仿真与烧录，则可取用其所产生16进制文件（*.hex）。

另一方面，我们可同时在**Altium Designer**的电路设计环境里，设计电路图、电路板，并产生制作电路板所需要的文件以及相关报告。

1.5.3 Keil μVision 3整合开发环境

Keil公司的μVision 3整合式开发环境（**I**ntegrated **D**evelopment **E**nvironment，简称**IDE**）是一套相当好用的8051开发软件，在整合式开发环境里，包括项目管理器（Project Manager）、源程序编辑器（Editor）、组译器（Assembler）、编译器（Compiler）、链接器（Linker/Locator）、调试器（Debugger）等，我们可从建立设计项目（Project）开始，然后编辑源程序（C语言或汇编语言）、编译、组译、链接，再进行调试，而调试就是一种程序功能仿真，如图1-24所示为其开发流程。

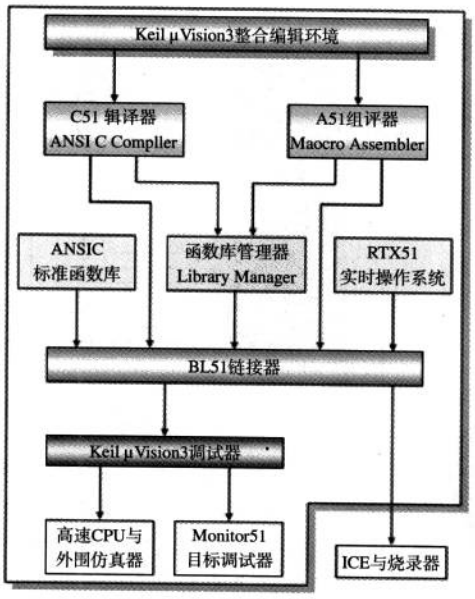


图1-24 Keil μVision 3开发流程

对于初次尝试Keil μVision 3的用户，Keil公司也慷慨地提供免费的评估版（evaluation version），用户使用满意再购买。当然，评估版也有些限制，就是无法生成超过2KB的可执行程序，但想要编写超过2KB的可执行程序也不是件简单的事！尤其对于初学者来说。本书中的范例，编译后产生的文件都小于2KB。所以，大家可放心使用这套开发环境。

1.5.4 89S51的在线烧录功能

究竟有什么理由能让89S51取代89C51成为下一代MCS-51的新主流产品？最有说服力的莫过于它所提供的在线烧录功能（**In-System Programmable**，简称**ISP**）。实际上，89C51也具有ISP的功能，而89S51的ISP功能更成熟，从此我们几乎可以跟IC烧录器说再见了！甚至也不一定需要昂贵的在线仿真器！

Atmel公司所提供的ISP电路与程序，可让用户通过个人计算机的并列端口或串行口（RS232C）直接将可执行文件下载到89C51/89S51。如此一来，不但可以省掉不少开发工具与设备，也非常适合初学者使用！

1.6 实例演练



在单片机系统的设计中，软件与硬件息息相关，不同的电路设计决定了不同的程序。因此，在编写程序之前，必须确定电路的连接状态。例如要利用8051的Port 2来控制8个LED，让这8个LED分成两组（高4位与低4位）交替闪烁，其设计步骤如下。



首先把电路连接妥当，如图1-25所示。

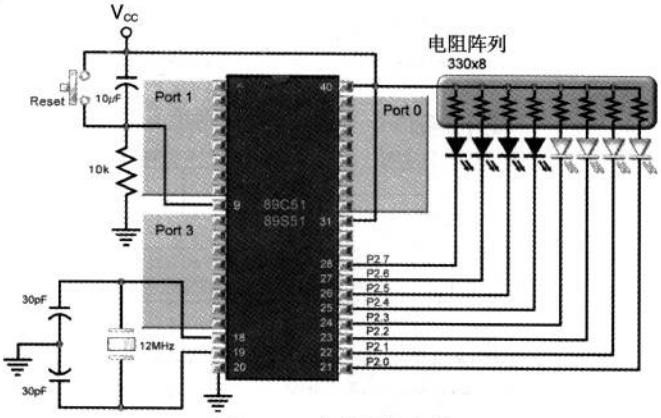


图1-25 交替闪烁电路

如图1-25所示，当Port 2的引脚输出低电压（0）时，所连接的LED呈现顺序偏压而发亮；若将引脚输出高电压（1）时，所连接的LED呈现逆序偏压而不亮。因此，我们的程序设计就要让Port 2输出为“00001111”以16进制数字表示为“0f”，使左边4个LED亮、右边4个LED不亮；而在Keil C的程序里16进制数字表示是以“0x”开头。所以，在程序里应表示为“0x0f”。隔一段时间后，再将输出反相（在Keil C里可利用“~”命令），即左边4个LED不亮、右边4个LED亮……，如此循环。



有了电路，又有了思路，随即将思路画成流程图，如图1-26所示，其中的延迟函数只是一个“0~x-1”的计数程序而已。

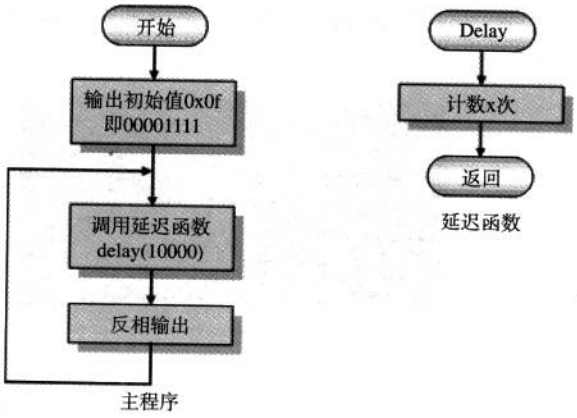


图1-26 流程图

Step 3 除非是非常简单的程序，否则，比较容易且保险的方法还是要根据流程图来编写程序！如下即为本实例的程序。

```
/* ch01.c - LED高低位交替闪烁程序 */
//==声明区=============================================
#include        <reg51.h>       // 定义8051寄存器的头文件,P2-17~19
#define LED     P2              // 定义LED接至Port 2
void delay(int);                // 声明延迟函数
//==主程序=============================================
main()                          // 主程序开始
{   LED=0x0f;                   // 初值=0000 1111,状态为左4个亮,右4个灭(共阳)
    while(1)                    // 无穷循环,程序一直运行
    {       delay(10000);       // 调用延迟函数
            LED=~LED;           // LED反相输出
    }                           // while循环结束
}                               // 主程序结束
//==子程序=============================================
void delay(int x)               // 延迟函数开始,x=延迟次数
{   int i;                      // 声明整数变量i
    for (i=0;i<x;i++);          // 计数x次
}                               // 延迟函数结束
```

ch01.c

Step 4 按 开始 钮（以Windows XP为例），在随即打出的菜单里，选中“所有程序”/“**Keil uVision3**”选项，即可运行Keil C，如图1-27所示。当然，若桌面上有 图标，可用鼠标双击这个图标一样可以进入Keil μVision 3环境。

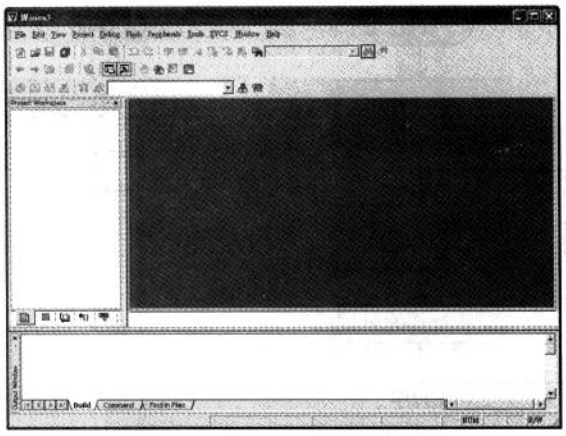


图1-27 Keil μVision 3主界面

Step 5 首先打开一个项目，运行**Project**菜单下的“**New Project...**”命令，屏幕上出现如图1-28所示的对话框。

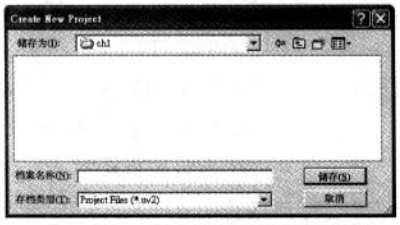


图1-28 Create New Project对话框

Step 6 在文件名称字段中指定所要新增的项目名称，再单击 储存(S) 按钮，屏幕上出现如图1-29所示的对话框。

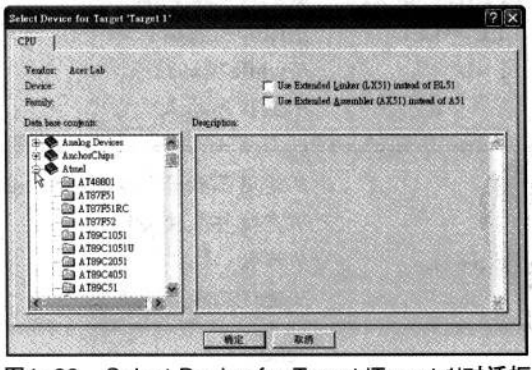


图1-29 Select Device for Target 'Target 1'对话框

Step 7 紧接着在**Data base contents**区域中，选中所要使用的CPU芯片（例如Atmel半导体公司的AT89S51），再单击 确定 按钮关闭对话框，屏幕上出现图1-30所示的对话框。

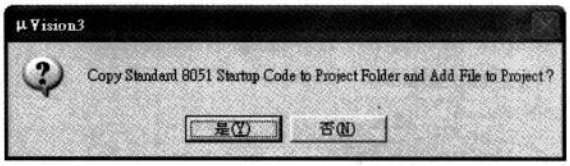


图1-30 µVision 3对话框

Step 8 这是询问我们要不要将8051启动码放入编辑的项目文件夹里。在此单击 是(Y) 按钮关闭此对话框，则左边区域中，将产生“**Target 1**”项目，如图1-31所示。

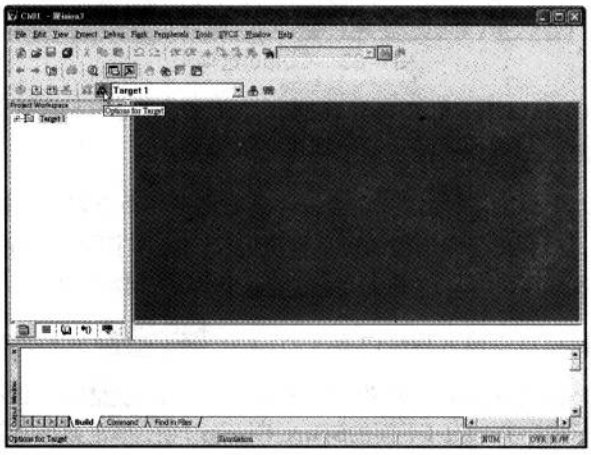


图1-31 Target 1项目

Step 9 单击 按钮设置此芯片的选项，屏幕上出现如图1-32所示的对话框。

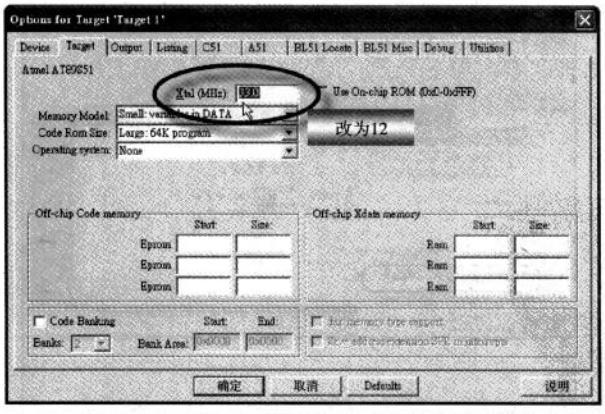


图1-32 Options for Target 'Target1'对话框

Step 10 在这个对话框里要设置此芯片的工作频率与所要输出的文件。首先在**Target**页的**Xtal（MHz）**字段输入**12**，选定此芯片的工作频率为12MHz。切换到**Output**选项卡，如图1-33所示。

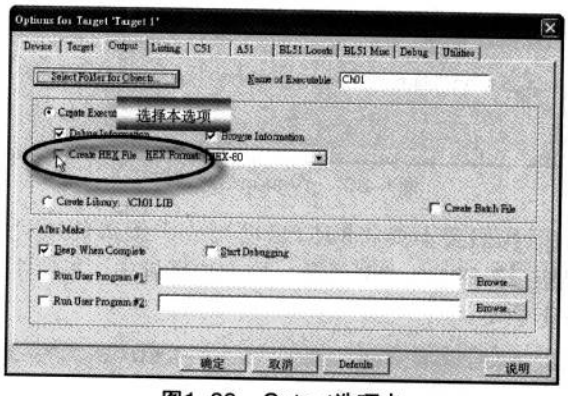


图1-33 Output选项卡

Step 11 选中**Create HEX File**复选框，如此才会产生16进制文件（*.hex），单击 确定 按钮关闭对话框即可完成设置。

Step 12 单击左上方的按钮，在编辑区里将打开一个全新的编辑窗口，再单击按钮，然后在随即出现的对话框里的文件名称字段中，输入指定所要保存的文件名称（ch01.c），再单击 储存(S) 按钮关闭对话框。

Step 13 在编辑窗口中输入程序内容（即1.6节 Step 3 中的ch01.c），其中排版的缩进是按 Tab 键所产生的，不要按一堆 键！程序编辑完成后，再将鼠标指向**Target 1**下面的**Sourc Group 1**项，单击鼠标右键，出现下拉菜单，如图1-34所示。选中**Add Files to Group Source Group 1**项，然后在随即出现的对话框里选定刚才编辑的ch01.c文件，再单击 Add 按钮；最后，单击 Close 按钮关闭对话框，即可将ch01.c文件加入Source Group 1。

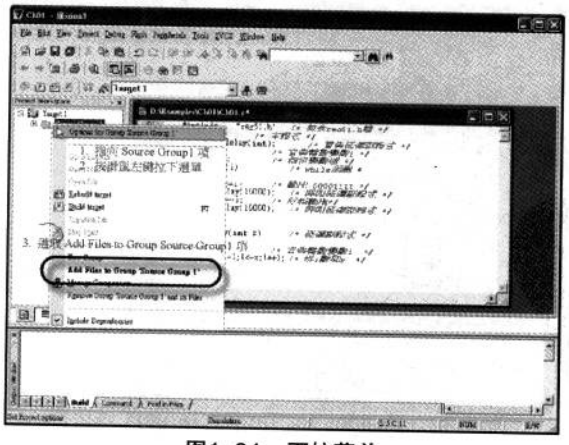


图1-34 下拉菜单

Step 14 紧接着进行编译与链接，单击左上方的按钮即可进行编译与链接，而其过程将记录在下方的输出窗口，如图1-35所示。

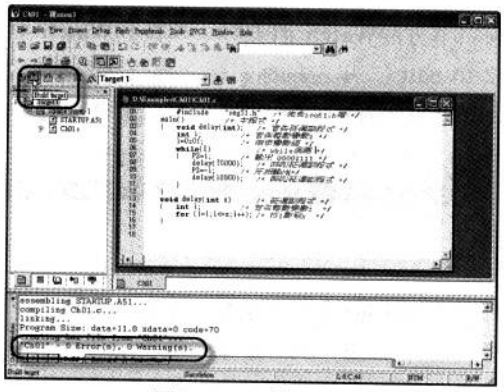


图1-35 过程记录

Step 15 “**0 Error(s), 0 Warnning(s).**”表示没有错误，因此就可继续进行调试/仿真。单击按钮开启调试工具栏，屏幕上出现确认对话框，如图1-36所示。

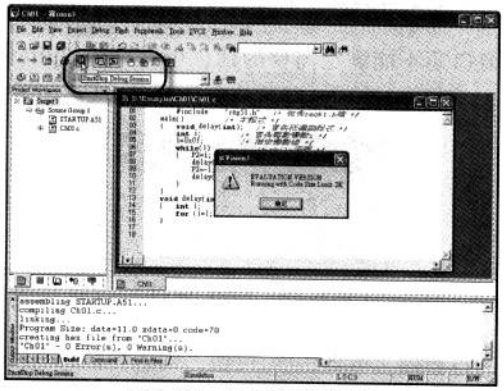


图1-36 开始调试/仿真

Step 16 单击 确定 按钮关闭对话框，即进入调试状态。若左下方没有出现监视窗口，可单击按钮打开它，如图1-37所示。

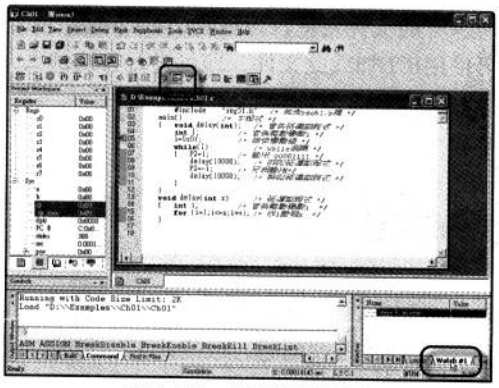


图1-37 打开监视窗口

Step 17 单击监视窗口下方的**Watch #1**卷标，切换到**Watch #1**页。指向**Name**字段里的<type F2 to edit>，单击鼠标左键，再按 F2 键，即可输入所要监视的信号名称，在此要追踪P2，所以输出P2，如图1-38所示。

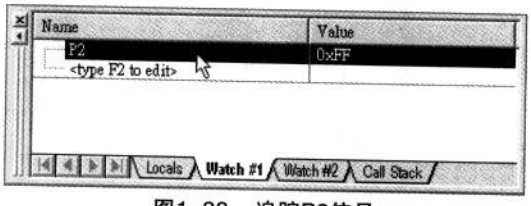


图1-38 追踪P2信号

Step 18 运行**View**菜单下的**Periodic Window Update**命令（若已打勾就不用再运行它），让窗口随程序进行而变动。再单击按钮，即开始执行程序。监视窗口中，P2的值也在0x0F与0xF0之间交替变化。表示连接在P2的LED将分为高四位与低四位交替闪烁。若要停止程序的进行，可单击按钮。

Step 19 如果监视窗口还不能满足需要的话，可运行**Peripherals**菜单中的**I/O-Ports**命令，再选中**Port 2**选项，即可打开Port 2窗口，如图1-39所示。

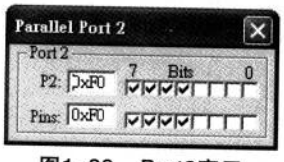


图1-39 Port2窗口

Step 20 若要继续进行程序，可单击按钮，则监视窗口与**Port 2**窗口的内容都将随程序的进行而变化。若想从头开始，则可单击按钮停止程序，单击按钮复位CPU，再单击按钮。若要关闭该项目，则先单击按钮离开调试状态，再运行**Project**菜单下的**Close Project**命令。这时候，只要运行**File**菜单下的**Exit**命令，即可关闭Keil C程序。

Step 21 在本项目所保存的文件夹里，可找到**Ch01.hex**文件，这个文件就是可执行文件。我们可使用ICE加载此文件，即可进行在线仿真。关于在线仿真的操作，可参考附录A。

1.7 实时练习



本章介绍了8051的基本知识，包括基本的硬件以及简单开发工具，这些都是学习8051的必备知识。在此请试着回答下列问题，以确认可顺利进入8051的世界。

选择题

（ ）1. 89S51的内部程序存储器与数据存储器的容量各为多少？
（A）64KB、128 B （B）4KB、64KB （C）4KB、128 B （D）8KB、256 B

（ ）2. 下列哪个是89S51比89C51多出的功能？
（A）存储器加倍 （B）.具有WDT功能
（C）多一个8位输入/输出端口 （D）多一个串行口

（ ）3. 在DIP40封装的8051芯片里，复位RESET引脚的引脚编号是多少？
（A）9 （B）19 （C）29 （D）39

（ ）4. 在DIP40封装的8051芯片里，接地引脚与电源引脚的引脚编号是多少？
（A）1、21 （B）11、31 （C）20、40 （D）19、39

（ ）5. 下列哪个软件同时提供8051的汇编语言及C语言的编译器？
（A）Keil μVision 3 （B）Java C++ （C）Dephi （D）Visual C++

（ ）6. 在12MHz时钟脉冲的8051系统里，一个机器周期有多长？
（A）1ms （B）12μs （C）1μs （D）12ms

（ ）7. 在8051芯片里，哪个引脚是控制使用内部程序存储器，还是外部程序存储器？
（A）XTAL1 （B）$\overline{EA}$ （C）$\overline{PSEN}$ （D）ALE

（ ）8. 下列哪个不是8051所提供的寻址模式？
（A）寄存器寻址（B）间接寻址（C）直接寻址（D）独立寻址

（ ）9. 下列哪个寄存器是8051内的16位寄存器？
（A）ACC （B）C （C）PC （D）R7

（ ）10. 开发微型计算机所使用的在线仿真器的简称是什么？
（A）ISP （B）USP （C）ICE （D）SPI

问答题

1. 试简述微型计算机的基本结构。

2. 试述在微型计算机里所使用的存储器可分为哪两大类？它们的用途分别是什么？

3. 试简述单片机与一般微型计算机的不同，单片机有何优点？

4. 试简述8051的基本结构以及89S51与89C51的不同。

5. 试简述8051的“位寻址”。

6. 试简述目前常见8051的3种封装方式，哪一种封装适用于学校或实验室？哪一种封装适合批量生产？

7. 试说明双列直插式8051单片机的各个引脚的名称与功能。

8. 试设计一个能让8051正常工作的基本电路。

9. 试说明哪些编号的MCS-51单片机内部不具备ROM？哪些具备EEPROM？

10. 试说明从外观上如何分辨内部具备EPROM与EEPROM的MCS-51单片机？

11. 在8051电路里，若要使用外部程序存储器，应如何连接？而访问外部数据存储器，

必须使用哪个指令？

12. 试述8051内部有多少个寄存器组？如何切换使用寄存器组？

13. 试简述PSW的意义，并说明其中各位的功能。

14. 试说明什么是SFR？它的地址在哪里？

15. 试说明什么是DPTR？它的功能是什么？

16. 试说明“位寻址”，哪里的存储器可位寻址？哪些特殊功能寄存器可位寻址？

17. 在12MHz的8051系统里，一个机器周期包括多少个状态周期？而一个状态周期是由几个时钟脉冲所组成的？

18. 试简介MCS-51程序的开发流程与开发中用到的工具？

第2章 认识μVision 3与Keil C

本章内容丰富，主要包括以下两部分。

μVision 3部分：

快速了解μVision整合设计环境。

Keil C语言部分：

认识Keil C语言的基本结构。

认识Keil C的变量、常数与数据类型。

认识存储器形式与工作模式。

认识Keil C的运算符、控制流程、函数与中断子程序。

认识Keil C的数组与指针。

认识Keil C的预处理命令。

2.1 μVision 3环境简介



与以往的版本相比，Keil μVision 3提供了更好的C语言或汇编语言的开发环境，下面介绍Keil μVision 3的开发环境。

2.1.1 认识μVision 3环境

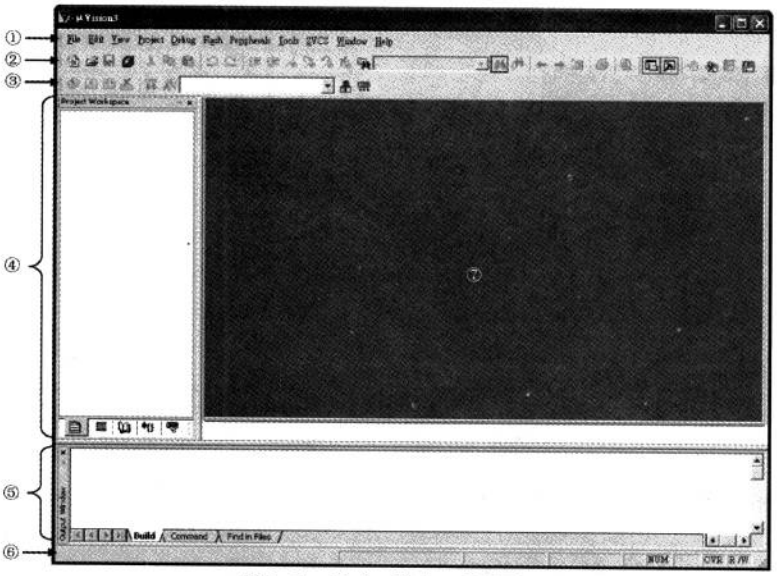


图2-1 Keil μVision 3窗口

总地来说，Keil μVision 3提供C语言与汇编语言的编辑、编译与链接、调试与仿真等功能，当然，还能生成在线仿真或烧录到芯片所需要的HEX文件等。当Keil μVision 3安装完成后，桌面上会出现一个 图标，只要双击该图标即可进入“Keil μVision 3”。若找不到这个图示，可按 按钮（以Windows XP为例），在打开的菜单中选择“所有程序”/“**Keil μVision3**”选项，同样可以进入Keil μVision 3，如图2-1所示。当然，为了说明方便，在图2-1里将Keil μVision 3窗口稍微缩小了一点，根据屏幕分辨率的不同以及窗口大小的不同，其中的工具栏会随之改变位置，对其中各项说明如下。

① 菜单栏：Keil μVision 3的菜单比以往版本的菜单要多，包括11个菜单，说明如下。

- **File**菜单提供个别文件的操作，如打开新文件（**New...**命令）、打开文件（**Open...**命令）、关闭文件（**Close**命令）、保存文件（**Save**命令）、另存为新文件（**Save As...**命令）等，这些文件操作命令大多可在文件工具栏里找到相对应的按钮。另外，我们可利用其最后一个命令（**Exit**命令）来关闭整个程序。
- **Edit**菜单提供编辑命令，如Windows软件都有的剪贴功能（**Copy**、**Cut**、**Paste**命令）、撤销/重复功能（**Undo**、**Redo**命令），还有文本处理程序的缩进/撤销缩进功能（**Indent Selected Text**、**Unindent Selected Text**命令）、书签功能（**Toggle Bookmark**、

Goto Next Bookmark、**Goto Previous Bookmark**、**Clear All Bookmarks**命令）、查找与替换功能（**Find...**、**Replace...**、**Find in Files...**、**Incremental Find**命令）等。

- **View**菜单提供窗口组件的显示开关，例如**Status Bar**命令用以切换是否显示状态栏，**File Toolbar**命令用以切换是否显示文件工具栏，**Build Toolbar**命令用以切换是否显示组建工具栏，**Debug Toolbar**命令用以切换是否显示调试/仿真工具栏，**Project Window**命令用以切换是否显示项目窗口，**Output Window**命令用以切换是否显示输出窗口，**Source Browser**命令用以切换是否显示源文件浏览器等。
- **Project**菜单提供项目管理功能，若要打开新项目，可使用**New Project...**命令；若要导入μVision 1版的项目，可使用**Import μVision1 Project...**命令；若要打开项目，可使用**Open Project...**命令；若要关闭项目，可使用**Close Project**命令等。
- **Debug**菜单提供调试/仿真的操作命令，不过当我们要执行调试/仿真时，大多直接操作调试/仿真工具栏上的按钮，而很少使用该菜单内的命令。
- **Flash**菜单提供芯片的下载与清零功能，也就是将可执行代码烧录到芯片中，也可将芯片中的数据清除。
- **Peripherals**菜单提供切换是否显示CPU内部各装置的显示窗口，如输出/输入端口等。
- **Tools**菜单提供PC-Lint程序语法检查工具。
- **SVCS**菜单提供版本管理功能。
- **Window**菜单提供工作区内的窗口排列功能。
- **Help**菜单提供帮助说明功能，其中包括多项通过Internet的帮助服务。

② 文件工具栏：Keil μVision 3将常用的功能放在该工具栏里，包括文件操作、剪贴功能、撤销与重复等与一般Windows软件类似的功能。而这些功能的按钮图案也与其他Windows软件类似，一看就懂！若有疑问，只要将鼠标指向该按钮，稍微停留一下，即出现小提示，如图2-2所示。

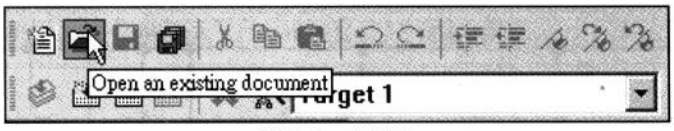


图2-2 小提示

此外，在此工具栏里还提供查找字符串的按钮（ ）、打印的按钮（ ）、开关调试/仿真工具栏的按钮（ ）、开关项目窗口的按钮（ ）、开关输出窗口的按钮（ ）等，可以说是主要的操作工具栏。

③ 组建工具栏：在Keil μVision 3里，8051程序的开发分为两个阶段，第一个阶段是程序编辑与组建（Build），所谓组建是指程序的编译/链接，及生成可执行文件。第二个阶段是调试/仿真，以确认程序的正确性。在第一阶段时，将打开该工具栏，可单击其中的 按钮进行选项设置，单击 按钮即可进行组建，单击 按钮进行重新组建。若该工具栏消失，可在**View**菜单下选择**Build Toolbar**命令，即可重新打开它。

④ 项目窗口：在左边长条型的项目窗口下方有5个图标，可将该窗口切换到不同的页面，如图2-3所示，其中各项说明如下。



图2-3 项目窗口的图标栏

- 鼠标指针指向**File**图标后单击左键，即可切换到文件管理页面，其中将列出项目里的所有文件。
- 鼠标指针指向**Regs**图标后单击左键，即可切换到寄存器页面，其中将列出CPU里的所有寄存器内容，以帮助我们进行调试工作；当然，即使我们不切换到该页面，进入调试/仿真状态时，也将自动切换到该页面。
- 鼠标指针指向**Books**图标后单击左键，即可切换到帮助说明页面，其中将列出所有说明项目。
- 鼠标指针指向**Functions**图标后单击左键，即可切换到函数页面，其中将列出所有函数。
- 鼠标指针指向**Templates**图标后单击左键，即可切换到样板页面，其中将列出所有样板。

我们可以单击按钮或选择**View**菜单下的**Project Window**命令，以切换是否显示项目窗口。

⑤ 输出窗口：在μVision 3窗口下方为输出窗口，其中包括3个页面，在**Build**页面里，将记录组建的过程与状况。在**Command**页面里，将记录操作的命令。在**Find in Files**页面里，将记录指定文件搜寻的结果。我们可以单击按钮或选择**View**菜单下的**Output Window**命令，以切换是否显示输出窗口。

⑥ 状态栏：在状态栏里包括7个字段，如图2-4所示，对其中各项说明如下。

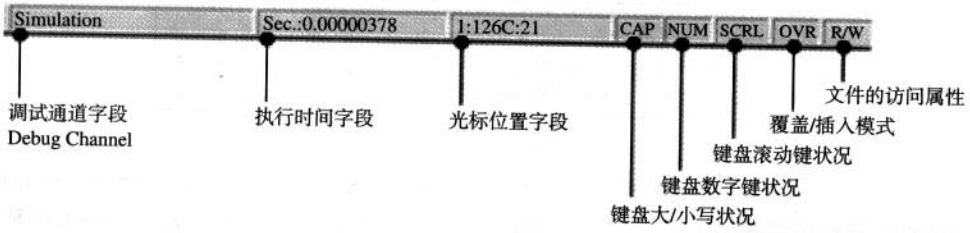


图2-4 状态栏

- 调试通道（Debug Channel）字段显示动作的调试工具。若使用内部的μVision仿真器，还会显示高级图形接口驱动程序（Advanced GDI Driver）或仿真（Simulation）。
- 执行时间字段显示执行仿真的时间。把鼠标指向此字段后单击右键，即可设置时序分析的标记。
- 光标位置字段显示光标所在位置。其中L表示第几行，C表示第几个字。

下列字段表示编辑器与键盘的状态：

- **CAP**表示目前键盘是锁住为大写状态（Caps Lock）；
- **NUM**表示目前键盘是锁住为数字键状态（Num Lock）；

- **SCRL**表示目前键盘是锁住为滚动状态（Scroll Lock）；
- **OVR**表示目前键盘输入模式为覆盖模式，若不显示则为插入模式；
- **R/W**或**R/O**表示目前所编辑的文件的属性，“**R/W**”表示该文件可读取与写入、“**R/O**”表示该文件为只读文件。

可以通过选择**View**菜单下的**Status Bar**命令来切换是否显示状态栏。

⑦ 工作区：在Keil μVision 3窗口中间一大片灰色区域为工作区，我们所打开/编辑的文件，将以窗口的形式出现在此区之中。若同时打开了多个文件，则可利用**Window**菜单里的命令，进行窗口排列的操作。

2.1.2 项目管理与选项

大部分的设计都是采用项目（Project）管理，在Keil μVision 3里也是采用项目管理。所有设计的开始都源自于新项目的建立或打开既有的项目。若要新建项目，可选择**Project**菜单下的**New Project...**命令；若要打开指定的项目，可选择**Project**菜单下的**Open Project...**命令。对一个新建的项目，除了项目窗口里多出一个**Target 1**项，工作区里是空的。除了上述操作，我们还得进行几个操作，说明如下。

加入源程序文件

若要将源程序文件加入目前的项目，可在项目窗口里，将鼠标指针指向**Target 1**下面的**Source Group 1**项，单击鼠标右键弹出菜单，选中其中的**Add Files to Group ‘Source Group 1’**选项，即可在随即打开的对话框里选择所要加入该项目的源程序文件，再单击 Add 按钮即可。我们可继续选择所要加入的文件，一个项目里可包含多个文件。最后，单击 Close 按钮关闭该对话框。

项目选项设置

项目选项的设置是一件重要的工作，单击 按钮即可打开选项对话框，如图2-5所示。

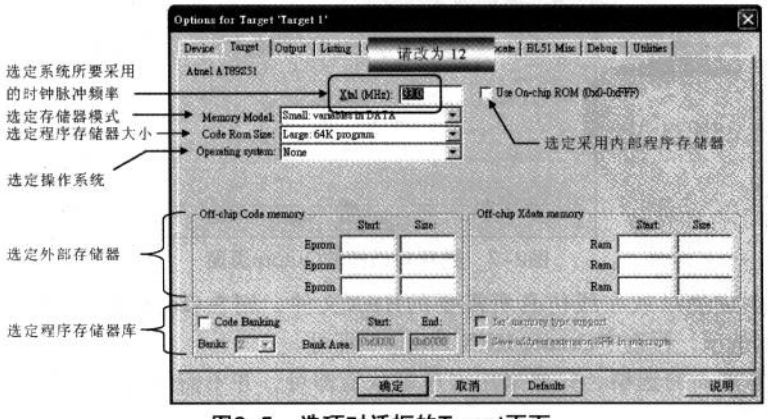


图2-5 选项对话框的Target页面

如图2-5所示，我们可在其中设置程序存储器等。其中的时钟脉冲频率的设置（即**Xtal**字段）与设计息息相关。它的默认值为该芯片的最高时钟脉冲频率。但在实际的电路里，并不一定使用最高频率，而是使用有助于程序设计与电路控制的频率，这个频率通常是12MHz（在此输入**12**即可）。另外，若要使用内部程序存储器，则选中**Use On-Chip ROM**选项。关于C语言的程序存储器规范，稍后会进行说明。若要改变或选择使用其他芯片，可将鼠标指针指向上方的**Device**图标，单击左键即可切换到**Device**页面，如图2-6所示。

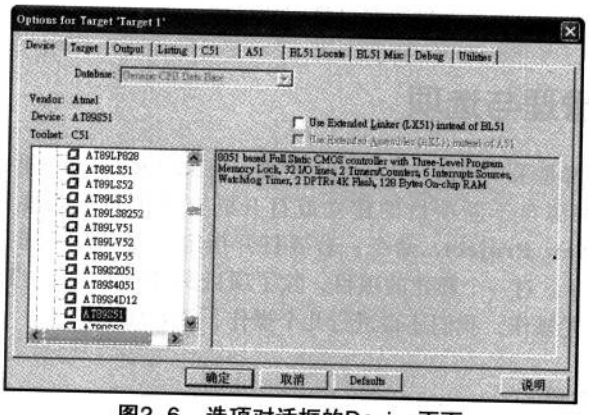


图2-6　选项对话框的Device页面

这时候就可在左边的字段中选择所要采用的芯片，其中这些字段的排列方式是按半导体厂商分类，这里列出了每个半导体厂商所提供的芯片。选择了所采用的芯片后，该芯片的说明将出现在右边区域之中。再将鼠标指针指向上方的**Output**图标，单击左键即可切换到**Output**页面，如图2-7所示。

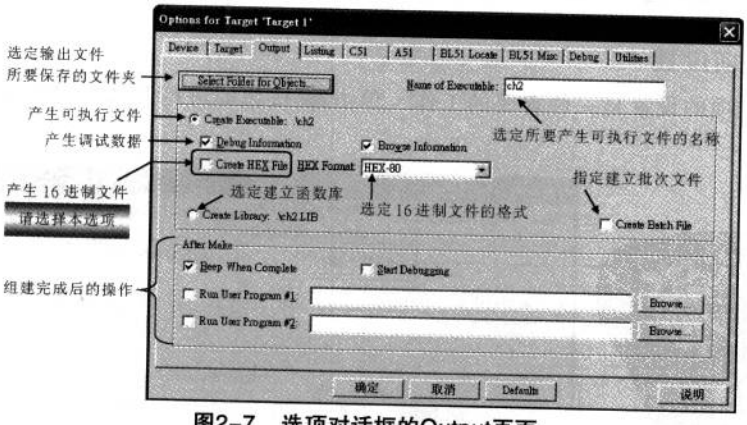


图2-7　选项对话框的Output页面

若要生成烧录或在线仿真所需要的HEX文件，则在**Output**页面里选中**Create Executable**选项，再选中**Create HEX File**选项即可。

其他各页面的选项，只要采用程序默认值即可，并不需要另行设置，其功能简述如下：

- **Listing**页面的功能是产生条列文件的相关设置；
- **C51**页面的功能是设置C51编译器的相关选项；
- **A51**页面的功能是设置A51组译器的相关选项；
- **BL51 Locate**页面的功能是设置BL51链接器的定位选项；
- **BL51 Misc**页面的功能是设置BL51链接器的其他选项；
- **Debug**页面的功能是设置调试器的相关选项；
- **Utilities**页面的功能是设置公用工具的相关选项。

当上述工作设置完成，且源程序编辑完成后，即可单击按钮进行组建。而程序的编写难免有错，若有错误，则在组建的过程中就会在下方的输出窗口中输出相关的错误信息，如图2-8所示。

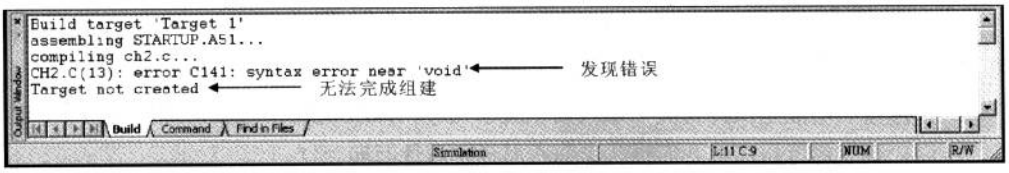


图2-8 组建过程中有错误

其中“CH2.C(13):error C141:...”表示程序的第13行有问题，我们可直接查看并修改它，然后单击按钮重新组建。若程序语法正确，则可通过组建过程，通过组建后将显示如图2-9所示的信息。

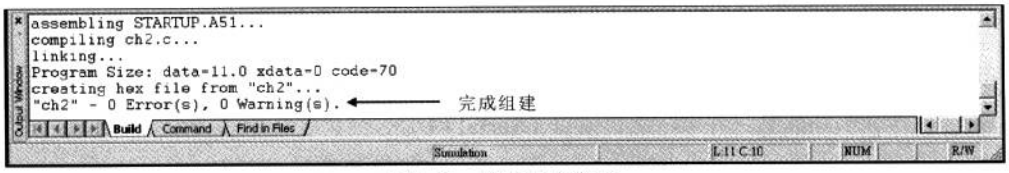


图2-9 成功完成组建

2.1.3 认识调试/仿真环境

完成组建后，可单击按钮进入调试/仿真状态，若使用试用版，屏幕上将出现如图2-10所示的确认对话框。



图2-10 确认对话框

这个对话框告诉我们目前使用的是试用版，限制了可执行程序不能超过2KB。当然，对于大部分的用户而言，2KB已经够大了！若要开发超过2KB的商用程序，那就请购买使用商用版。单击 确定 按钮关闭对话框，这时窗口里的组建工具栏不见了，取而代之的是调试/仿真工具栏，如图2-11所示，其中各按钮说明如图2-11所示。



图2-11 调试/仿真工具栏

该按钮的功能是复位（Reset）CPU，同时让程序从头开始执行。

该按钮的功能是全速执行程序。

该按钮的功能是停止程序的执行。

该按钮的功能是单步执行，每单击一下执行一个指令，若遇到子程序（函数），则跳入该子程序（函数），同样一步一步执行子程序（函数）里的指令。

该按钮的功能是单步执行，每单击一下执行一个指令，若遇到子程序（函数），则直接执行完成该子程序（函数）。

该按钮的功能是完成当时所执行的子程序（函数），跳出该子程序（函数），返回主程序。

该按钮的功能是执行到文字插入点（文字光标，即I形光标）所在的那一行指令，所以，在按该按钮之前，先将文字插入点移至选择的那一行。

该按钮的功能是打开**STARUP.A51**窗口，以展示程序相对应的汇编语言执行状况。

该按钮的功能是打开/停止跟踪记录。

该按钮的功能是显示跟踪记录，并打开反组译窗口（**Disassembly**）。

该按钮的功能是打开反组译窗口（**Disassembly**）。

该按钮的功能是打开监视窗口（**Watch**），在右下方。

该按钮的功能是打开指令码包含率窗口（**Code Coverage**）。

该按钮的功能是打开串行端口窗口（**Serial #1**）。

该按钮的功能是打开存储器窗口（**Memory**），它在屏幕的中下方。

该按钮的功能是打开效率分析器窗口（**Prefermance Analyzer**）。

该按钮的功能是打开逻辑分析仪窗口（**Logic Analyzer**），以进行时序（波形）分析。

该按钮的功能是打开符号窗口（**Symbols Analyzer**），与项目窗口同在一个位置。

该按钮的功能是打开工具箱窗口（**Toolbox**），工具箱窗口用来在进行调试分析时更新显示。

2.1.4 外围操作

在调试/仿真状态下，**Peripherals**菜单的内容较丰富，而且对于调试工作有不小的帮助！如图2-12所示。

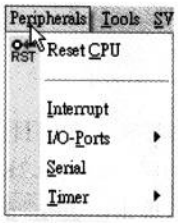


图2-12 调试/仿真状态下的Peripherals菜单

对Peripherals菜单中的各项命令说明如下。

Reset CPU命令

该命令的功能是复位CPU，与单击RST按钮的功能一样。

Interrupt命令

该命令的功能是切换是否显示中断系统对话框（**Interrupt System**），如图2-13所示。

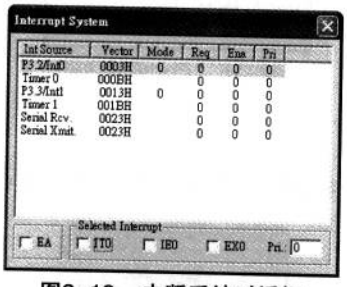


图2-13 中断系统对话框

其中列出该芯片中的所有中断源，直接选中所要操作的中断源，则该中断源的所有相关选项将呈现于对话框下方。以选中P3.2/Int0选项为例，对话框下方将出现下列选项。

- **EA**选项表示程序是否设置打开中断源的总开关。若选中了此选项，表示程序中设置打开中断源的总开关。
- **IT0**选项表示程序所设置的中断触发方式。若选中了此选项（即IT0=1），表示该中断采用边缘触发方式；若没有选中此选项（即IT0=0），表示该中断采用低电平触发方式。
- **IE0**选项为触发该中断的信号。若要触发该中断，可将鼠标指向该选项，单击左键则此选项将闪一下，随即进入执行其中断子程序。
- **EX0**选项表示程序是否处于该中断源的中断状态中。若选中了此选项（即EX0=1），表示程序已处于该中断之中。
- **Pri**字段表示程序对该中断所设置的优先等级。

取复位CPU，与单击RST按钮的功能一样。

I/O-Ports命令

该命令的功能是切换是否显示输出入端口对话框（**Parallel Port**），而选中该命令后，将出现输入/输出端口下拉菜单，如图2-14所示。

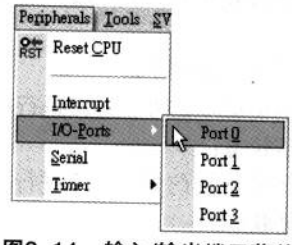


图2-14 输入/输出端口菜单

以选中**Port0**选项为例，将打开如图2-15所示的对话框，其中分为**P0**与**Pins**两行，**P0**行显示该输出入端口的输出状况，**Pins**行则为输入状况，我们可在此行中输入信号，其中打勾为1，没有打勾为0。

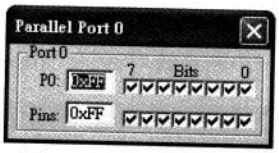


图2-15 Port 0对话框

Serial命令

该命令的功能是切换是否显示串行端口对话框（Serial Channel），如图2-16所示。

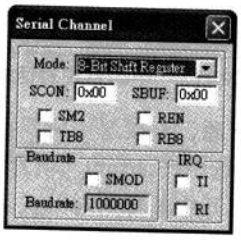


图2-16 Serial Channel对话框

其中各项说明如下：

- **Mode**字段为程序中所设置的串行端口模式；
- **SCON**字段为程序中所设置的**SCON**寄存器的内容；
- **SBUF**字段为串行口寄存器（**SBUF**）的内容；
- **SM2**选项为程序中所设置的**SM2**位的状态，若选中此选项表示SM2=1，不选中此选项表示SM2=0；
- **REN**选项为程序中所设置的**REN**位的状态，若选中此选项表示REN=1，不选中此选项表示REN=0；
- **TB8**选项为程序中所设置的**TB8**位的状态，若选中此选项表示TB8=1，不选中此选项表示TB8=0；
- **RB8**选项为程序中所设置的**RB8**位的状态，若选中此选项表示RB8=1，不选中此选项表示RB8=0；
- **SMOD**选项为程序中所设置的**SMOD**位的状态，若选中此选项表示SMOD=1，不选中此选项表示SMOD=0；
- **Baudrate**字段为程序中所设置的传输率；
- **TI**选项为串行端口传出中断的触发信号，程序进行时，将鼠标指向此选项，单击左键即可进入串行口传出中断状态；
- **RI**选项为串行端口接收中断的触发信号，程序进行时，将鼠标指向此选项，单击左键即可进入串行口接收中断状态。

Timer命令

该命令的功能是切换是否显示计时计数对话框（**Timer**），而选中该命令后，将出现计时计数器下拉菜单，如图2-17所示。

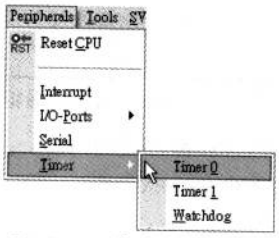


图2-17 计时计数器菜单

以**Timer 0**选项为例，将打开如图2-18所示的对话框，其中各项说明如下。

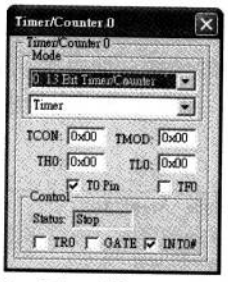


图2-18 Timer/Counter 0对话框

- **Mode**区域里包括两个字段，上面字段为程序中所设置的计时计数器模式，下面字段为程序中所设置的是内部计时（**Timer**选项），还是外部计数（**Counter**选项）。
- **TCON**字段为程序中所设置**TCON**寄存器的内容。
- **TMOD**字段为程序中所设置**TMOD**寄存器的内容。
- **TH0**字段为程序中所设置计量寄存器（**TH0**）的内容。
- **TL0**字段为程序中所设置计量寄存器（**TL0**）的内容。
- **T0 Pin**选项为芯片T0引脚（P3.4）的状态。若选中此选项表示T0引脚为高电平；不选中此选项表示T0引脚为低电平。
- **TF0**选项为计时计数器中断标志。若此选项为选中状态，表示TF0=1，目前为计时计数器中断状态；此选项没有选中，表示TF0=0，目前没有进入计时计数器中断状态。
- **Status**字段表示目前是否启用计时计数功能。
- **TR0**选项的功能是软件运行计时计数器。若此选项为选中状态，表示TR0=1，即软件运行计时计数器；此选项没有选中，表示TR0=0，即停用计时计数器。
- **Gate**选项为程序所设置的控制开关状态。若此选项为选中状态，表示Gate=1，即由外部运行计时计数器；此选项没有选中，表示Gate=0，即由内部运行计时计数器。
- **INT0#**选项为外部运行计时计数器的引脚。当设置为外部运行计时计数器的状态下，将鼠标指向此选项，单击左键即可运行计时计数器。而INT0#表示P3.2引脚，

INT1#表示P3.3引脚。

再以**Watchdog**选项为例，将打开如图2-19所示的对话框，其中各项说明如下。

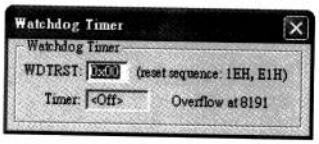


图2-19 Watchdog对话框

- **WDTRST**字段表示目前WDTRST寄存器的内容。
- **Timer**字段表示目前看门狗定时器的状态。

2.2 Keil C语言的基本结构



总地来说，C语言的程序可看作是由一堆函数（Function，或视为子程序）所构成的，其中的主程序是以“main()”开始的函数，而每个函数可看作独立的个体，就像是模块（Module）一样。所以，C语言是一种非常模块化的程序语言。用C语言所设计的程序的基本结构如图2-20所示。

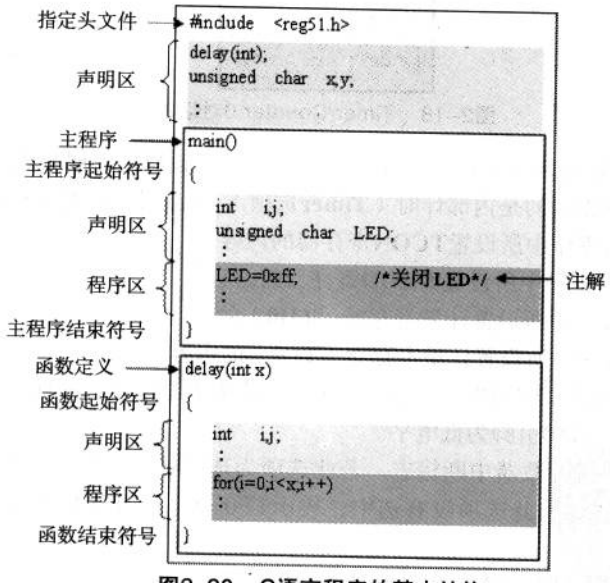


图2-20 C语言程序的基本结构

对其中各项说明如下。

指定头文件

“头文件”或称为包含文件（*.h），这是一种将预先定义好的基本数据，放入此程序之中。在8051程序里，必要的头文件是定义8051内部寄存器地址的数据，如下所示。

```
/*--------------------------------------------------------------------------
REG51.H
Header file for generic 80C51 and 80C31 microcontroller.
Copyright (c) 1988-2002 Keil Elektronik GmbH and Keil Software, Inc.
All rights reserved.
--------------------------------------------------------------------------*/


#ifndef__REG51_H__
#define__REG51_H__

/*  BYTE Register  */
sfr P0   = 0x80;                    /* Port 0 */
sfr P1   = 0x90;                    /* Port 1 */
sfr P2   = 0xA0;                    /* Port 2 */
sfr P3   = 0xB0;                    /* Port 3 */
sfr PSW  = 0xD0;                    /* 程序状态字组 */
sfr ACC  = 0xE0;                    /* A累积器 */
sfr B    = 0xF0;                    /* B寄存器 */
sfr SP   = 0x81;                    /* 堆栈指针寄存器 */
sfr DPL  = 0x82;                    /* 数据指针寄存器的低8位 */
sfr DPH  = 0x83;                    /* 数据指针寄存器的高8位 */
sfr PCON = 0x87;                    /* PCON寄存器 */
sfr TCON = 0x88;                    /* TCON寄存器 */
sfr TMOD = 0x89;                    /* TMOD寄存器 */
sfr TL0  = 0x8A;                    /* Timer0计量寄存器的低8位 */
sfr TL1  = 0x8B;                    /* Timer1计量寄存器的低8位 */
sfr TH0  = 0x8C;                    /* Timer0计量寄存器的高8位 */
sfr TH1  = 0x8D;                    /* Timer1计量寄存器的高8位 */
sfr IE   = 0xA8;                    /* IE寄存器 */
sfr IP   = 0xB8;                    /* IP寄存器 */
sfr SCON = 0x98;                    /* SCON寄存器 */
sfr SBUF = 0x99;                    /* SBUF寄存器 */

/*  BIT Register  */
/*  PSW   */
sbit CY   = 0xD7;                   /* 进位位 */
sbit AC   = 0xD6;                   /* 辅助进位位 */
```

```
sbit F0  = 0xD5;                    /* 用户标志位 */
sbit RS1 = 0xD4;                    /* 寄存器组选择位1 */
sbit RS0 = 0xD3;                    /* 寄存器组选择位0 */
sbit OV  = 0xD2;                    /* 溢出位 */
sbit P   = 0xD0;                    /* 同位位 */

/* TCON */
sbit TF1 = 0x8F;                    /* Timer 1的溢出位 */
sbit TR1 = 0x8E;                    /* Timer 1的运行位 */
sbit TF0 = 0x8D;                    /* Timer 0的溢出位 */
sbit TR0 = 0x8C;                    /* Timer 0的运行位 */
sbit IE1 = 0x8B;                    /* INT1的中断标志 */
sbit IT1 = 0x8A;                    /* INT1的触发信号种类位 */
sbit IE0 = 0x89;                    /* INT0的中断标志*/
sbit IT0 = 0x88;                    /* INT0的触发信号种类位 */

/* IE */
sbit EA  = 0xAF;                    /* 中断的总开关 */
sbit ES  = 0xAC;                    /* 串行端口中断的启用位 */
sbit ET1 = 0xAB;                    /* Timer1中断的启用位 */
sbit EX1 = 0xAA;                    /* INT1中断的启用位 */
sbit ET0 = 0xA9;                    /* Timer0中断的启用位 */
sbit EX0 = 0xA8;                    /* INT0中断的启用位 */

/* IP */
sbit PS  = 0xBC;                    /* 串行端口中断优先等级设置位 */
sbit PT1 = 0xBB;                    /* Timer0中断优先等级设置位 */
sbit PX1 = 0xBA;                    /* INT1中断优先等级设置位 */
sbit PT0 = 0xB9;                    /* Timer1中断优先等级设置位 */
sbit PX0 = 0xB8;                    /* INT0中断优先等级设置位 */

/* P3 */
sbit RD  = 0xB7;                    /* RD引脚 */
sbit WR  = 0xB6;                    /* WR引脚 */
sbit T1  = 0xB5;                    /* T1引脚 */
sbit T0  = 0xB4;                    /* T0引脚 */
sbit INT1 = 0xB3;                   /* INT1引脚 */
sbit INT0 = 0xB2;                   /* INT0引脚 */
```

```
sbit TXD  = 0xB1;                    /* TxD引脚 */
sbit RXD  = 0xB0;                    /* RxD引脚 */

/*  SCON  */
sbit SM0  = 0x9F;                    /* 串行端口模式设置位0 */
sbit SM1  = 0x9E;                    /* 串行端口模式设置位1 */
sbit SM2  = 0x9D;                    /* 串行端口模式设置位2 */
sbit REN  = 0x9C;                    /* 接收脉冲控制位 */
sbit TB8  = 0x9B;                    /* 传送的bit 8位 */
sbit RB8  = 0x9A;                    /* 接收的bit 8位 */
sbit TI   = 0x99;                    /* 传送的中断标志 */
sbit RI   = 0x98;                    /* 接收的中断标志 */

#endif
```

指定头文件的方式有如下两种。

- 在#include之后，以<>包含头文件的文件名，如下所示。若采用这种方式，编译程序将从**Keil μVision3**的头文件的文件夹查找所指定的头文件。如果Keil μVision3安装在C盘根目录下，则编译程序将从C:\Keil\C51\INC路径中查找它。

```
#include <头文件的文件名>
```

- 在#include之后，以" "包含头文件的文件名，如下所示。若采用这种方式，编译程序将从源程序所在文件夹里查找所选择的头文件。

```
#include "头文件的文件名"
```

声明区

在选择头文件之后，可声明程序之中所使用的常数、变量、函数等，其作用域将扩展到整个程序，包括主程序与所有函数，所以要特别小心！不过，在此建议，若程序之中使用到函数，则可在此先声明所有使用到的函数，这样，函数定义的先后次序将不会有所影响。换言之，在引用该函数的程序之前或之后都可以进行函数定义。若没有在此声明函数，则在使用一个函数之前，必须得先定义该函数！

主程序

如图2-21所示，主程序是以main()为开头，整个内容放置在一对大括号（即{}）里。其中分为声明区与程序区，在声明区里所声明的常数、变量等，仅适用于主程序中，而不影响其他函数。若在主程序之中使用了某函数，但在之前的声明区中没有声明，也可在主程序的声明区中声明。另外，程序区放的就是指令所构成的程序的内容。

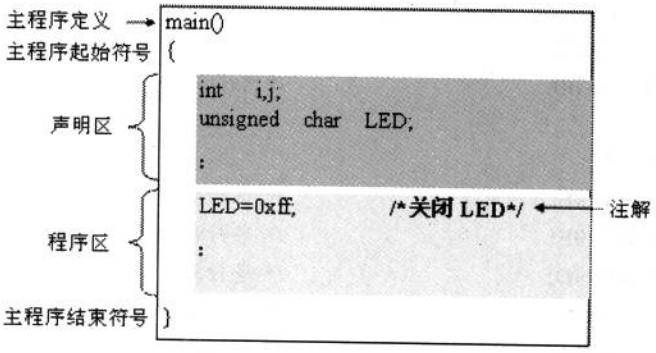


图2-21 主程序的结构

函数定义

函数是一种具有相对独立的功能的程序，其结构与主程序类似。不过，函数可将所要处理的数据传入该函数里，称为传入自变量（Arguments）；也可将函数处理完成后的结果传回调用它的程序，称为返回值。不管是传入自变量还是返回值，在定义函数的第一行里要交待清楚！其格式如下：

返回值的数据形式 **函数名称号**（传入自变量的数据形式）

例如要将一个无符号字符（unsigned char）传入函数，而函数执行完成时，要传回一个整数（int），若此函数的名称为My_func，则其函数定义为：

```
int My_func(unsigned char x)
```

若不要传入函数，则可在小括号内选择为void。同样地，若不要返回值，则可在函数名称左边指定为void，或根本不指定。另外，函数的起始符号、结束符号、声明区及程序区都与主程序一样。在一个C语言的程序里，可使用多个函数。

注解

所谓“注解”就是说明，属于编译器不处理的部分。C语言的注解是以“/*”开始，以“*/”结束，注解可以跟在指令之后，也可以在独立的一行中。注解可以使用中文，不过μVision 3对中文的处理并不是很好，常会造成文字定位不准确等麻烦，所以应尽量避免使用中文注解。另外，注解也可以用“//”开头，该符号右边整行都是注解。

2.3 变量、常数与数据类型



在C语言中，常数（constant）与变量（variables）都是为某个数据指定的存储器位置，其中常数是在指定存储器位置里，存放固定不变的数据，而变量是在指定存储器位置里的数据是可变的。声明常数或变量的格式如下：

数据类型 常数/变量名称[=初始值];

其中的［=初始值］并非必需的项目，分号（;）是结束符号，例如要声明一个整数类型的x变量，其初始值为50，则使用如下指令：

```
int x = 50;
```

若不要初始值，则使用如下指令：

```
int x;
```

若要同时声明x、y、z三个整数类型的变量，则变量名称之间，以“,”分隔，则使用如下指令：

```
int x,y,z;
```

2.3.1 数据类型

既然常数或变量的声明是由编译程序为该常数或变量保留存储器位置，那么就要交代清楚保留多大的位置？这与常数或变量的数据类型有关。在声明常数或变量的格式中，一开始就要指明数据类型，可见数据类型的重要性！在Keil C所提供的数据类型可分为下列几类。

● 通用数据类型

通用数据类型可用于一般C语言之中，如ANSI C等，包括字符（char）、整数（int）、浮点数（float）与无（void），其中字符与整数又分为有符号（signed）与无符号（unsigned）两类（使用小写），通用数据类型如表2-1所示。

表2-1

型 态	名 称	位 数	范 围
char	字符	8	-128~+127
unsigned char	无符号数字元	8	0~255
enum	列举	8/16	-128~+127/-32 768~+32 767
short	短整数	16	-32 768~+32 767
unsigned short	无符号整数	16	0~65 535
int	整数	16	-32 768~+32 767
unsigned int	无符号整数	16	0~65 535
long	长整数	32	-2^{31}~$2^{31}-1$
unsigned long	无符号长整数	32	0~$2^{32}-1$
float	浮点数	32	$\pm 1.175\,494 \times 10^{-38}$~$3.402\,823 \times 10^{38}$
double	双倍精度浮点数	32	$\pm 1.7 \times 10^{308}$
void	无	0	无

8051的特殊数据类型

针对8051硬件装置所设置的数据类型有4种，分别是bit、sbit、sfr及sfr16，如表2-2所示。

表2-2

名称	位数	范围
bit	1	0、1
sbit	1	0、1
sfr	8	0~255
sfr16	16	0~65 535

以下将特别介绍这4种8051特殊数据类型（使用小写）。

- bit数据类型是定义1个位的变量，将会被存放在0x20~0x2f之间的地址。
- 通常sbit数据类型是用于访问内部可位寻址的数据存储器，即0x20~0x2f之间的存储区。或访问可位寻址的特殊功能寄存器（SFR），即0x80~0xff之间的存储区。若要使用sbit数据形式，则其声明方式有下列几种。
 - 先声明一个bdata存储器形式（存储器形式稍后介绍）的变量，再声明属于该变量的sbit变量，如下所示。

```
char bdata scan;      /* 声明scan为bdata存储器类型的字符*/
sbit input_0=scan^0; /* 声明input_0为scan变量的bit0*/
```

当我们要指定（声明）某个变量的第n位，则可在该变量名称右边加上“^n”即可。例如P0的bit 3用P0^3表示。

 - 先声明一个sfr变量（稍后介绍），再声明属于该变量的sbit变量，如下所示。

```
sfr P0=0x80;          /* 声明P0为0x80存储器位置，即Port 0*/
sbit P0_0=P0^0;       /* 声明P0_0为P0变量的bit0*/
```

这种用法最方便，因为8051内部特殊功能寄存器的声明都在reg51.h里（如表2-1所示）。而程序的开头已经将这个文件包含进来了，所以我们只要声明sbit即可，如以上的第二行。

 - 直接指定存储器位置，例如要声明P0的bit 0，则使用如下指令：

```
sbit P0_0=0x80^0;     /* 声明P0_0为0x80地址的bit0*/
```

不过，这种用法要求我们必须熟记每个地址才行。

- 通常sfr数据类型是用于8051内部特殊功能寄存器（寄存器名称使用大写），即0x80~0xff地址，与内部存储器的地址相同。不过，特殊功能寄存器与内部存储区是独立的，我们必须以不同的访问方式来区分。特殊功能寄存器采用直接寻址方式访问，而内部存储器采用间接寻址方式访问。在Keil C里，所谓直接寻址就是直接指定其地址，以Port 0的声明为例，直接寻址的指令如下：

```
sfr P0=0x80;          /* 声明P0为0x80存储器位置，即Port 0*/
```

所谓间接寻址，就是声明为idata存储器形式（存储器形式稍后介绍）的变量使用如下指令。

```
char idata BCD;       /* 声明BCD变量为间接寻址的存储器位置 */
```

由于reg51.h里已声明8051内部特殊功能寄存器，我们不必要再次进行声明，而用户又不想亲自动手配置存储器（交给编译程序处理），所以程序里很少出现sfr数据类型的声明。

▶ 通常sfr16数据类型用于8051内部16位的特殊功能寄存器（寄存器名称使用大写），如Timer 2的捕获寄存器（RCAP2L、RCAP2H），Timer 2的计量寄存器（TL2、TH2），数据指针寄存器（DPL、DPH）等。以数据指针寄存器为例，使用如下指令。

```
sfr16 DPTR=0x82;    /* 声明DP变量为数据指针寄存器*/
```

2.3.2 变量名称与保留字

由上述声明常数或变量的格式中可知，在数据类型之后就是变量名称。变量名称除了从方便理解的角度来设定外，还要遵守下列规则：

- 可使用大/小写字母、数字或下划线（即_）；
- 第一个字符不可以是数字；
- 不能使用保留字。

所谓“保留字”是指编译程序将该字符串保留为其他特殊用途，ANSI C的保留字，如表2-3所示。

表2-3

asm	auto	break	case	char	const
continue	default	do	double	else	entry
enum	extern	float	for	fortran	goto
int	long	register	return	short	signed
sizeof	static	struct	switch	typedef	union
undigned	void	volatile	while		

当然，Keil C也有其特有的保留字，如表2-4所示。

表2-4

at	_priority_	_task_	alien	bdata	bit
code	compact	data	far	idata	interrupt
large	pdata	reentrant	sbit	sfr	sfr16
small	using	xdata			

2.3.3 变量的作用范围

变量的作用范围或称作用域，与该变量在哪里声明有关，大致可分为如下两种。

● 全局变量

若在程序开头的声明区或者说是没有大括号限制的声明区，所声明的变量的作用域为整个程序，称为全局变量（Global Variables）。如图2-22所示，其中的LED、SPEAKER就是全局变量。

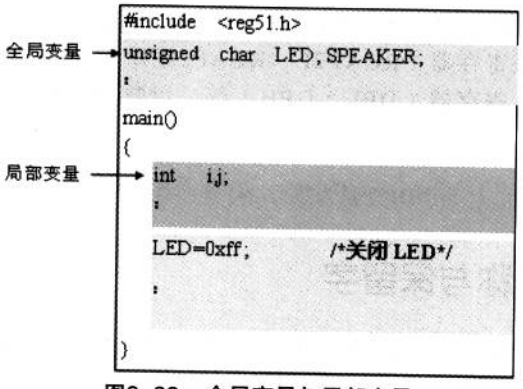


图2-22 全局变量与局部变量

● 局部变量

若在大括号内的声明区所声明的变量，其作用域将受限于大括号，称为局部变量（Local Variables）。如图2-22中的i、j就是局部变量。若在主程序与各函数中都声明了相同名称的变量，则离开主程序或函数，该变量将自动消失，又称之为自动变量（Automatic Variables）。如图2-23所示，在主程序与delay子程序中，各声明了变量i、j，但主程序的i、j，与delay子程序中的i、j是各自独立的，它们之间没有关系。

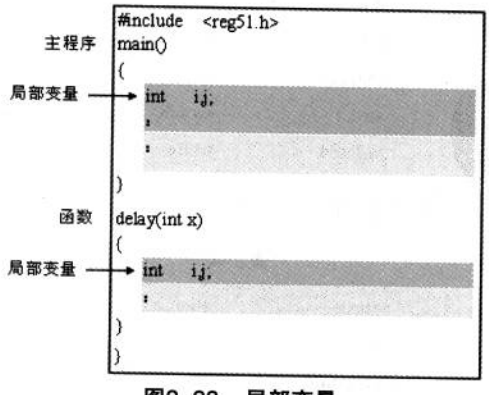


图2-23 局部变量

2.4 存储器形式与模式



8051的程序设计属于硬件驱动程序的设计，所以与8051内部结构息息相关，特别是存储器。在本节中将介绍8051的存储器形式与其工作模式。

2.4.1 存储器形式

Keil C对于存储器的管理是将存储器分成6种形式，如表2-5所示。

表2-5

存储器形式	说 明	适用范围
code	程序存储器	0x0000~0xffff(64k)
data	直接寻址的内部数据存储器	0x00~0x7f(128)
idata	间接寻址的内部数据存储器	0x80~0xff(128)
bdata	位寻址的内部数据存储器	0x20~0x2f(16)
xdata	以DPTR寻址的外部数据存储器	64KB之内
pdata	以R0、R1寻址的外部数据存储器	256 B之内
far	扩展的ROM或RAM外部存储器，仅适用于少数的芯片，如Philips 80C51MX、Dallas 390等	最大可达16MB

对各种形式说明如下（使用小写）。

● 程序存储器

code存储器形式为8051的程序存储器。在第1章中曾经提及，标准的8051内部有4KB程序存储器，可扩展至64KB；8052内部有8KB程序存储器，可扩展至64KB。而新版或兼容的51芯片，其内部的程序存储器有16KB的和32KB的，甚至还有64KB的。顾名思义，程序存储区就是用来存放程序代码的存储区，是一种只能读取不能写入的只读存储器。除了用来存放程序代码外，程序存储区也可存放固定的数据，例如7段LED数码管的驱动信号、LED数组的显示信号、音乐的驱动信号、LCM的显示字符串等。使用如下所示的指令能够实现用数组的方式（稍后介绍）储存表格：

```
char code SEG[10]={     0x03, 0x9f, 0x25, 0x0d, 0x99,
                        0x49, 0xc1, 0x1f, 0x01, 0x19 };
```

● 内部数据存储器

在第1章中讲过，标准的8051内部有128 B数据存储器，可扩展至64KB；8052内部有256 B程序存储器，可扩展至64KB。而新版或兼容的51芯片，其内部的数据存储器有512 B、768 B等不同规格。由于8051内部的特殊功能寄存器与数据存储器的地址相同，必须采用不同的寻址方式才能区分出是访问特殊功能寄存器还是访问数据存储器。当然，对

于汇编语言而言，可以用不同的指令来区分直接寻址与间接寻址。不过，Keil C并没有直接寻址与间接寻址的指令，但可以用不同的存储器形式来区分操作的对象，它有data、idata及bdata三种存储器形式。其中data存储器形式可直接存取0x00~0x7f数据存储器，例如指定x为字符类型的变量：

```
char data x;
```

idata存储器形式可间接寻址方式访问0x80~0xff数据存储器，其声明方式如下：

```
char idata x;
```

bdata存储器形式可位寻址方式访问0x20~0x2f数据存储器，其声明方式如下：

```
bit bdata x;
```

● 外部数据存储器

对于外部存储器的访问方式，汇编语言提供专用的指令（即movx），而Keil C并没有特别为存取外部存储器提供指令，所以必须以存储器形式来区分，若要声明存取64KB范围外部存储器的字符变量，声明方式如下：

```
char xdata x;
```

若要声明存取256 B范围外部存储器的字符变量，声明方式如下：

```
char pdata x;
```

2.4.2 存储器模式

Keil C提供**SMALL**、**COMPACT**及**LARGE**三种存储器模式（Memory Models），用来决定未标明存储器形式的函数的自变量（Arguments）、自动变量及变量声明等的初始化存储器形式。对这三种存储器模式的说明如下。

- 小型模式（**SMALL**）将所有变量初始化为8051的内部存储器，其作用域就像在声明区里明确地声明data存储器形式一样。若指定为此种模式，变量的存取最有效率，对于用户而言无异是最佳的选择。
- 精简模式（**COMPACT**）将所有变量初始化为外部存储器的一页（Page），也就是256 B。而其寻址方式为高8位经过Port 2，我们必须在startup码中设置它，编译器并不会帮我们设置这个输入/输出端口。使用这种模式，就像在声明区里明确地声明pdata存储器形式一样。当然，在这种模式下，虽然变量的大小可达256 B，但其存取效率不如SMALL模式，比稍后要介绍的LARGE模式好些。
- 大型模式（**LARGE**）将所有变量初始化为外部存储器，其作用域就像在声明区里明确地声明xdata存储器形式一样。在这种模式下，虽然变量的大小可达64KB，但其存取效率比前面两种模式还差。

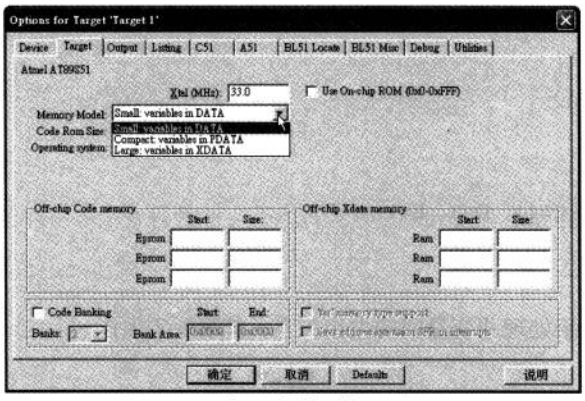


图2-24 选项对话框的Target页面

在μVision 3里，若要设置存储器模式，可单击 按钮打开选项对话框，如图2-24所示。这时候，就可在**Memory Model**字段中，选中所要采用的存储器模式。

2.5 Keil C的运算符



运算符（Operator）就是程序语句中的操作符号，Keil C的运算符可分为以下几种。

算术运算符

顾名思义，算术运算符就是执行算术运算的操作符号。除了一般人所熟悉的四则运算（加减乘除）外，还有取余数运算，如表2-6所示。

表2-6

符号	功能	范例	说明
+	加	A=x+y	将x与y变量的值相加，其和放入A变量
-	减	B=x-y	将x变量的值减去y变量的值，其差放入B变量
*	乘	C=x*y	将x与y变量的值相乘，其积放入C变量
/	除	D=x/y	将x变量的值除以y变量的值，其商数放入D变量
%	取余数	E=x%y	将x变量的值除以y变量的值，其余数放入E变量

▶ 程序范例

```
main()
{   int  A, B, C, D, E, x, y;
    x=7;
    y=2;
    A=x+y;
    B=x-y;
    C=x*y;
```

```
    D=x/y;
    E=x%y;
}
```

▶ 程序结果

```
A=9、B=5、C=14、D=3、E=1
```

关系运算符

关系运算符用于处理两变量间的大小关系，如表2-7所示。

表2-7

符号	功能	范例	说明
==	相等	x==y	比较x与y变量的值是否相等，相等则其结果为1，不相等则为0
!=	不相等	x!=y	比较x与y变量的值是否相等，不相等则其结果为1，相等则为0
>	大于	x>y	若x变量的值大于y变量的值，其结果为1，否则为0
<	小于	x<y	若x变量的值小于y变量的值，其结果为1，否则为0
>=	大等于	x>=y	若x变量的值大于或等于y变量的值，其结果为1，否则为0
<=	小等于	x<=y	若x变量的值小于或等于y变量的值，其结果为1，否则为0

▶ 程序范例

```
main()
{   int A, B, C, D, E, F, x, y;
    x=7;
    y=2;
    A=(x==y);
    B=(x!=y);
    C=(x>y);
    D=(x<y);
    E=(x>=y);
    F=(x<=y);
}
```

▶ 程序结果

```
A=0、B=1、C=1、D=0、E=1、F=0
```

逻辑运算符

逻辑运算符就是执行逻辑运算功能的操作符号。逻辑运算包括AND（及）、OR（或）、NOT（反相），其结果为1或0，如表2-8所示。

表2-8

符号	功能	范例	说明
&&	及运算	(x>y)&&(y>z)	若x变量的值大于y变量的值，且y变量的值也大于z变量的值，其结果为1，否则为0
\|\|	或运算	(x>y)\|\|(y>z)	若x变量的值大于y变量的值，或y变量的值也大于z变量的值，其结果为1，否则为0
!	反相运算	!(x>y)	若x变量的值大于y变量的值，则其结果为0，否则为1

▶ 程序范例

```
main()
{   int  A, B, C, x, y, z;
    x=7;
    y=2;
    z=5;
    A=(x>y)&&(y<z);
    B=(x==y)||(y<=z);
    C=!(x>z);
}
```

▶ 程序结果

A=1、B=1、C=0

布尔位运算符

布尔位运算符与逻辑运算符非常相似，它们之间最大的差异在于布尔位运算符针对变量中的每一个位，逻辑运算符则是对整个变量进行操作。如图2-25所示为布尔位运算符的运算示意图。

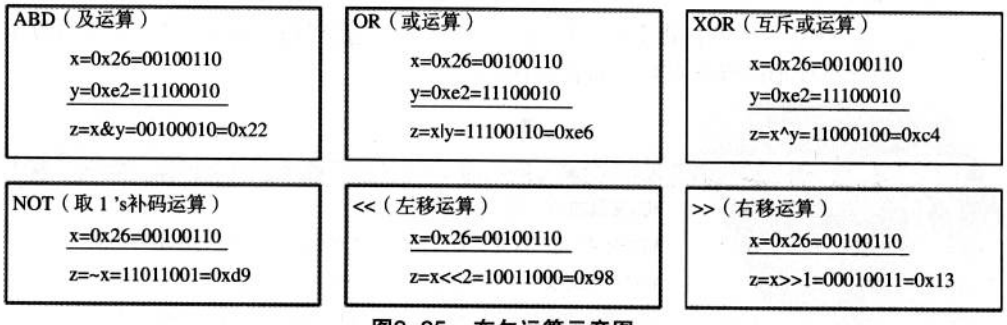


图2-25 布尔运算示意图

布尔运算的运算方式如表2-9所示。

表2-9

符号	功能	范例	说明
&	及运算	A=x&y	将x与y变量的每个位，进行AND运算，其结果放入A变量
\|	或运算	B=x\|y	将x与y变量的每个位，进行OR运算，其结果放入B变量
^	互斥或	C=x^y	将x与y变量的每个位，进行XOR运算，其结果放入C变量
~	取1's补码	D=~x	将x变量的值，进行NOT运算，其结果放入D变量
<<	左移	E=x<<*n*	将x变量的值左移*n*位，其结果放入E变量
>>	右移	F=x>>*n*	将x变量的值左移*n*位，其结果放入F变量

▶ 程序范例

```
main()
{   char  A, B, C, D, E, F, x, y;
    x=0x25; /* 即00100101 */
    y=0x73; /* 即01110011 */
    A=x&y;
    B=x|y;
    C=x^y;
    D=~x;
    E=x<<3;
    F=x>>4;
}
```

▶ 程序结果

```
A=0x21（即00100001）、B=0x77（即01110111）、C=0x56（即01010110）
D=0xda（即11011010）、E=0x28（即00101000）、F=0x02（即00000010）
```

● 赋值运算符

赋值运算符是一种很有效率而且特殊的操作符号，包括最直接的“=”，还有将算术运算、逻辑运算变形的操作符号，如表2-10所示。

表2-10

符号	功能	范例	说明
=	指定	A=x	将x变量的值，放入A变量
+=	加入	B+=x	将B变量的值与x变量的值相加，其和放入B变量，与B=B+x相同
-=	减去	C-=x	将C变量的值减去x变量的值，其差放入C变量，与C=C-x相同
=	乘入	D=x	将D变量的值与x变量的值相乘，其积放入D变量，与D=D*x相同
/=	除	E/=x	将E变量的值除以x变量的值，其商放入E变量，与E=E/x相同
%=	取余数	F%=x	将F变量的值除以x变量的值，其余数放入F变量，与F=F%x相同
&=	及运算	G&=x	将G变量的值与x变量的值进行AND运算，其结果放入G变量，与G=G&x相同

续表

符号	功能	范例	说明
\|=	或运算	H\|=x	将H变量的值与x变量的值进行OR运算，其结果放入H变量，与H=H\|x相同
^=	互斥或	I^=x	将I变量的值与x变量的值进行XOR运算，其结果放入I变量，与I=I^x相同
<<=	左移	J<<=*n*	将J变量的值左移*n*位，与J=J<<*n*相同
>>=	右移	K>>=*n*	将K变量的值右移*n*位，与K=K>>*n*相同

▶ 程序范例

```
main()
{   unsigned char   A=0x52, B=0x3a, C=0x01, D=0x15,
                    E=0xaa, F=0x11, G=0xf0, H=0x1f,
                    I=0x55, J=0x68, K=0x75, x=0x96;
    A=x;
    B+=x;           /* B=B+x */
    C-=x;           /* C=C-x */
    D*=x;           /* D=D*x */
    E/=x;           /* E=E/x */
    F%=x;           /* F=F%x */
    G&=x;           /* G=G&x */
    H|=x;           /* H=H|x */
    I^=x;           /* I=I^x */
    J<<=2;          /* J=J<<2 */
    K>>=3;          /* K=K>>3 */
}
```

▶ 程序结果

```
A=0x96、B=0xd0、C=0x6b、D=0x4e、E=0x01、F=0x11
G=0x90、H=0x9f、I=0xc3、J=0xb0、K=0x0e
```

● 递增/减运算符

递增/减运算符也是一种很有效率的运算符，其中包括递增与递减等两种操作符号，如表2-11所示。

表2-11

符号	功能	范例	说明
++	加1	x++	将x变量的值加1
--	减1	x--	将x变量的值减1

▶ 程序范例

```
main()
```

```
{   int     A, B;
    int     x=5,y=10;
    A=x++;
    B=y--;
}
```

▶ 程序结果

```
A=6、B=9
```

● 运算符的优先级

程序中的语句可能使用了不止一个运算符，而多个运算符放在一个语句之中的情况下，必须要有规则才不会弄错！就像日常生活里的加减乘除一样，我们很自然地遵守“先乘除后加减”的原则，所以“3×2+8÷2”应该是先进行“3×2”及“8÷2”，再把这两项操作的结果相加，大概不会有人搞错。总的来说，横式的操作都是按“由左到右”的顺序，除非遇到较高优先等级的运算符或操作符号，最常见的就是小括号，小括号内的操作先进行。例如“3×(2+8)÷2”，就得先操作“2+8”，再把结果放回去排队！由此可知，运算符或操作符号的优先等级很重要，如表2-12所示为Keil C运算符或操作符号的优先等级。

表2-12

优先级	运算符或操作符号	说明
1	(、)	小括号
2	~、!	取补码、反相运算
3	++、--	递增、递减
4	*、/、%	乘、除、取余数
5	+、-	加、减
6	<<、>>	左移、右移
7	<、>、<=、>=、==、!=	关系运算符
8	&	布尔运算 - AND
9	^	布尔运算 - XOR
10	\|	布尔运算 - OR
11	&&	逻辑运算 - AND
12	\|\|	逻辑运算 - OR
13	=、*=、/=、%=、+=、-=、<<=、>>=、&=、^=、\|=	赋值运算符

2.6 Keil C的流程控制



总的来说，程序的结构是由上而下逐行执行。不过，我们可以借助流程控制的指令与语句来改变程序流程，以达到省力并且能够根据需要跳转的效果让我们的程序更灵活。Keil C

所提供的流程控制指令与语句可区分为3种，即循环指令、选择指令及跳转指令，说明如下。

2.6.1 循环指令

循环指令就是将程序流程控制在指定的循环里，直到符合指定的条件才结束循环，继续往下执行。Keil C所提供的循环指令有for语句、while语句、do-while语句（使用小写），说明如下。

计数循环

for语句是一个很实用的计数循环，其格式如下：

```
for(表达式1; 表达式2; 表达式3)
{
        指令;
        [break]
        :
}
```

其中有3个表达式，说明如下。

- 表达式1为循环初始值，例如从0开始则写成“i=0;”，其中的i必须事先声明过。若之前没有声明，也可直接在此声明，例如将它声明为整型，即“int i=0;”。另外，其中的“;”是分隔符，不能省略！
- 表达式2为判断条件，它是执行循环的条件。例如“i<20;”，则只要满足i<20就继续执行循环。若此表达式空白，只有“;”，例如“for(i=0; ;i++)”或“for(;;)”，则会无条件执行循环，不会跳出循环。
- 表达式3为循环变量增值，最常见的是递增或递减，例如“i++”或“i--”，当然也可以运用其他运算方式，例如每次增加2，即“i+=2”。

▶ 使用范例1。

```
for(int i=0;i<8;i++)
说明：
循环执行8次
```

▶ 使用范例2：

```
for(int x=100;x>0;x--)
说明：
循环执行100次
```

▶ 使用范例3：

```
for(;;)
```

```
说明：
死循环
```

▶ 使用范例4：

```
for(int num=0;num<99;num+=5)
说明：
循环执行20次
```

for语句的下面，可用上一对大括号，大括号中包含了所要执行的指令，指令逐行写入。若该循环只要执行一行指令，可不使用大括号，例如要从0~9，将table数组中的数据依序输出到P2，语句如下。

```
for(i =0;i<10;i++)
        P2=table[i];
```

另外，若循环未达到跳出循环的条件，而其他条件成立，要强制跳出循环，则可在循环内加入其他条件与break指令，语句如下。

```
for(i =0;i<100;i++)
{       :
        if (sw1==0) break;
        :
}
```

前条件循环

while语句将判断条件放在语句之前，称为前条件循环，其格式如下。

```
while（表达式）
{
        指令;
        [break]
        :
}
```

前条件循环是指当其中的表达式成立时才开始执行其下大括号内的内容。例如i不等于0时，才执行循环，语句如下。

```
while(i !=0)
{       :
        指令;
        :
}
```

若while的表达式为1，则形成死循环，语句如下。

```
while(1)
{       :
        指令;
        :
}
```

同样地，若大括号内只有一行指令，则可省略大括号，语句如下。

```
while(i !=0)
        i--;
```

另外，若循环未达到跳出的条件，而其他条件成立，要强制跳出循环，则可在循环内加入其他条件与break指令，语句如下。

```
while(1)
{       :
        if (sw1==0) break;
        :
}
```

● 后条件循环

do-while语句提供先执行再判断的功能，称为后条件循环，其格式如下。

```
do      {
        指令;
        [break]
        :
        } while(表达式);
```

在这个语句里，将先执行一次循环后，再判断表达式是否成立。若不成立，则不会再执行该循环。例如i不等于0时，才执行循环，语句如下。

```
do      {
        指令;
        :
        } while(i !=0);
```

若while的表达式为1，则形成死循环，语句如下。

```
do      {
        指令;
        :
```

```
} while(1);
```

同样地，若大括号内只有一行指令，则可省略大括号，语句如下。

```
do i--; while(i !=0);
```

另外，若循环未达到跳出的条件，而其他条件成立，而要强制跳出循环，则可在循环内加入其他条件与break指令，语句如下。

```
do      {
        if (sw1==0) break;
        :
        } while(1);
```

2.6.2 选择指令

选择指令的功能是根据条件决定程序的流程，Keil C所提供的选择指令有if-else语句及switch-case语句（使用小写），说明如下。

● 条件选择

if-else语句提供条件判断的语句，称为条件选择语句，其格式如下：

```
if(表达式)
{
        指令区域1;
        :
}
else
{
        指令区域2;
        :
}
```

在这个语句里，将先判断表达式是否成立？若成立，则执行**指令区域1**。若不成立，则执行**指令区域2**，如图2-26所示。

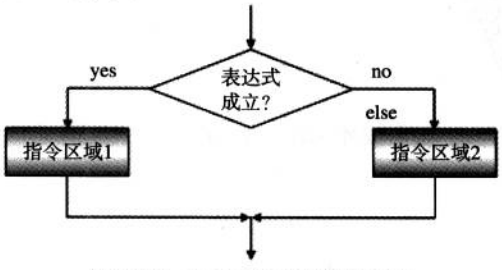


图2-26 if-else条件选择流程图

其中else部分也可省略，即写成如下格式，流程图如图2-27所示。

```
if(表达式)        {指令区域1;}
其他指令;
```

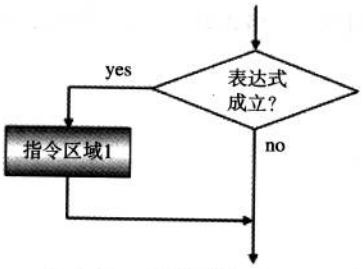


图2-27 if条件选择流程图

if-else语句也可利用else if指令串接为多重条件判断，即写成如下格式，流程图如图2-28所示。

```
if(表达式1)
        {指令区域1;      }
else if(表达式2)
        {指令区域2;      }
else if(表达式3)
        {指令区域3;      }
else    {指令区域4;      }
        :
```

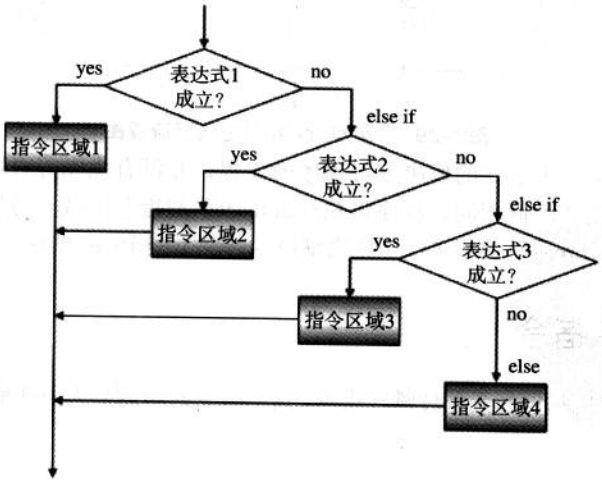


图2-28 if-else if条件选择流程图

在这种流程下，从表达式1开始判断，若表达式1成立，那么表达式2、表达式3都不起作用。同样地，若表达式1不成立，而表达式2成立，则表达式3不起作用。很明显，指

令区域1的优先等级最高，其次才是指令区域2、指令区域3或指令区域4。

多分支选择

switch-case语句提供多重选择，就像是波段开关一样，称为多分支选择语句。这种选择方式不会有优先等级的问题，其格式如下。流程图如图2-29所示。

```
switch(表达式)
{   case(常数1):
            { 指令区域1; }
            break;
    case(常数2):
            { 指令区域2; }
            break;
            :
    default:
            { 指令区域n; }
            break;
}
```

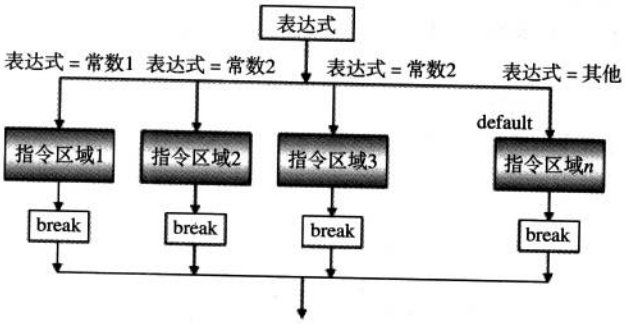


图2-29 switch-case多重选择流程图

在这种流程下，表达式的值决定可程序的流程，并没有优先等级之分。若没有一个分支的常数与表达式的值相同，程序将执行default下的指令区域。另外，在每个分支结束时必须有一个“break”的指令，否则会继续执行其下一个case的指令区域。

2.6.3 跳转指令

goto是Keil C所提供的无条件跳转指令。这个指令的功能是无条件地改变程序的流程，其格式如下：

```
goto 标号
```

这个指令与汇编语言的jmp指令一样，其右边是一个标号（label），当执行到这个指令后，将跳转到该标号相应的语句上，如下所示。

```
goto    loop;
        :
loop:   指令
        :
```

2.7 数组与指针



数组（array）是一种将同类型数据集合管理的数据结构，而指针（Pointer）是存放存储器地址的变量。因此，数组与指针可谓是数据管理的好帮手。

2.7.1 数组

总地来说，数组也是一种变量，将一堆相同数据形态的变量，以一个相同的变量名称来表示。既然是一种变量，使用之前就得声明，其声明格式如下：

数据类型 数组名 [数组大小];

如下所示的语句声明一个含有9个字符的数组：

```
char LCM[9];
```

那么这个数组包括LCM[0]~LCM[8]等9个字符，是字符的数组，相当于常说的“字符串”。但是Keil C没有“字符串”数据类型，所用字符数组来代替字符串。

声明变量是为该变量指定存储器位置。不同的数据类型，预留的存储器空间各不相同。例如声明一个字符变量，则预留1字节的存储器给该变量；声明一个整数变量，则预留2字节的存储器给该变量。具体细节我们可以参考表2-2。对于上述的LCM数组，程序将预留8字节的存储器给它。

声明数组的时候，也可以给它赋初始值，语句如下。

```
char LCM[9]="Testing.";
```

以上语句表示LCM[0]的初始化内容为**T**，LCM[1]的初始化内容为e，...LCM[8]的初始化内容为 .。但要注意，字符串的最后面必须以“\0”结束，所以需要9个字符来存放以上字符串。若不知道数组的大小可不填数组大小，而由程序来处理，语句如下。

```
char string1[]= “Welcome to China.” ;
```

如果声明整数（int）或浮点数（float）数组，也要指定其初始值，可利用大括号实现，语句如下。

```
int Num[6]={30, 21, 1, 45, 26, 37};
```

上面介绍的是一维数组，也可以声明多维数组，如下所示为声明*n*维数组的格式：

数据类型 数组名[数组大小1] [数组大小2]...[数组大小*n*];

以声明一个二维3 × 2整数数组为例，如下所示：

```
int Num[3][2]={{10,11}, {12,13}, {14,15}};
```

表示Num[0][0]的初始化内容为10，Num[0][1]的初始化内容为11，...Num[2][1]的初始化内容为15。

完成声明后，就可以像对一般变量一样对其进行操作，如下所示。

```
a=Num[0][1]+3;
```

执行以上语句后，a的值为14。

2.7.2 指针

指针是用来存放存储器地址的变量。既然是变量，使用前当然要声明，声明格式如下。

数据类型 *变量名称;

通常指针都采用整数数据类型，例如要声明一个名为ptr的指针，语句如下。

```
int *ptr;
```

也可以把同类型的变量与指针放在一起声明，语句如下。

```
int *ptr1, *ptr2, a, b, c;
```

与指针息息相关的运算符是“&”，这个运算符的功能是取得变量的地址。使用这个运算符可以将指定的变量的地址放入指针变量，以便后续操作，语句如下。

```
ptr1=&a;
```

执行以上语句后，a变量的地址就被放入ptr1指针变量。当然，这些操作主要是针对数组，通常会先取得数组中第一个元素的地址，语句如下。

```
ptr1=&Num[0][0];
```

执行以上语句后，Num数组的首地址将被放入ptr1指针变量。若要将Num[0][0]的内容输出到Port 2，语句如下。

```
P2=Num[0][0];
```

或以指针变量的方式实现以上操作，语句如下。

```
P2=*ptr1;
```

同理，若要将Num[1][1]的内容赋给Port 2，语句如下。

```
P2=Num[1][1];
```

或以指针变量的方式实现以上操作，语句如下。

```
P2=*(ptr1+3);
```

2.8 函数与中断子程序



总地来说，函数（Function）和中断子程序都属于子程序，也可以称函数为子程序，称中断子程序为中断函数，根据习惯的不同会有不同的称呼。

2.8.1 函数

函数的结构与主程序的结构类似，不过函数还能传入自变量，传出自变量，如图2-30所示为函数的结构。

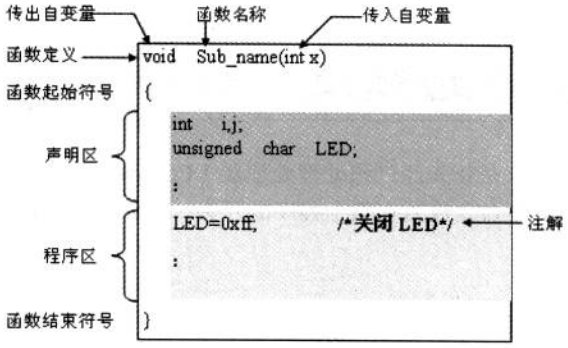


图2-30 C语言程序中的函数基本结构

函数是一种独立功能的程序。可将所要处理的数据传入该函数里，称为传入自变量。当然，可以传入不止一个自变量；另外，也可将函数处理完成后的结果传回调用它的程序，称为返回值。不管是传入自变量还是传入返回值，在定义函数的第一行中一定要交待清楚！

2.8.2 中断子程序

中断子程序的结构与函数的结构类似，不过中断子程序不能传入自变量和传出自变量。而且使用中断子程序之前不需要声明，但需要在主程序中进行中断的相关设置（将

在第6章进行详细介绍）。

从中断子程序的第一行就可看出它与一般函数的不同，如下所示：

```
void 中断子程序名称（void）interrupt中断编号 using 寄存器组
```

对其中各项说明如下：

- 由于中断子程序并不传入自变量，也不返回值，所以在其左边标识“void”，在中断子程序名称右边的括号里，也是“void”。
- 中断子程序的命名只要是符合规定的字符串即可。
- interrupt右边标示中断编号，Keil C提供0~31等32个中断编号，不过8051只使用0~4，8052则使用0~5。例如要声明为INT0外部中断，则标识为“interrupt 0”；若要声明为T0计时计数器中断，则标识为“interrupt 1”。
- using右边标示中断子程序里所要采用的寄存器组。8051内部有4组寄存器组，即RB0~RB3。通常主程序使用RB0，根据需要在子程序里使用其他寄存器组，以避免数据的冲突。若不指定寄存器组，则可省略本项目。

例如要定义一个INT0的中断子程序，其名称定义为“INT”，而在该中断子程序使用RB1寄存器组，则应声明为：

```
void INT(void) interrupt 0 using 1
```

然后在接下来的大括号内，编写中断子程序的内容。

2.9 Keil C的预处理命令



所谓“预处理命令”是指先经过预处理处理器（Pre-Processor）处理过后，才进行编译的命令。通常，预处理命令放置在整个程序的开头，除非是条件式编译命令。Keil C提供如下3种预处理命令。

定义命令

#define命令用来指定常数、字符串或宏函数的代名词，与汇编语言的“equ”、“reg”命令一样。#define命令的格式如下。

```
#define 代名词 常数(字符串或宏函数)
```

例如要从Port 2输出，则可将outputs定义为Port 2，语句如下。

```
#define outputs P2
```

进行了以上定义后，如果在程序中要输出到P2的指令，就可以用outputs代替，语句如下。

```
outputs = 0xff;        /* 输出11111111 */
```

进行编译时，预处理器会将整个程序里的所有“outputs”替换为“P2”，所以把这个指令改为。

```
P2 = 0xff;  /* 输出11111111 */
```

这样有什么好处呢？例如原先针对某个电路所设计的程序，电路原本是由P2输出的，但因某些因素，电路改为由P0输出，或其他电路不是由P2输出的电路，也想使用这个程序来驱动，怎么办？我们只要改这一行就好了，而不必在程序之中寻找所要改的地方，更不会有漏改的情况发生。当然，使用预处理命令也有助于程序的阅读与理解。另外，还可根据需要使用多行#deifne命令。

包含命令

#include命令的功能是将指定的定义或声明等文件放入程序之中。

条件式编译命令

C语言是一种可移植性很高的程序语言，源程序可在不同版本的C语言编译器下进行编译。当然，不同的C语言编译器提供不同的资源与指令语法。这时候，就可应用条件式编译命令来区分不同的编译器。在8051的程序设计里也可以应用条件式编译命令以适应不同的外围与控制方式。条件式编译命令的格式如下。

```
#if     表达式
        程序1
#else
        程序2
#endif
```

若表达式成立，则编译程序1，否则编译程序2。

2.10 实时练习



在本章中介绍了μVision 3环境及Keil C，这些都是学习8051的基本知识与必备技能。在此请试着回答下列问题，以确认可顺利学习8051的相关知识。



（ ）1. 在Keil μVision 3里开发8051程序的第一步是什么？
（A）打开新的项目文件（B）调试与仿真（C）组建程序（D）产生执行文件

（ ）2. 在Keil μVision 3里，若要打开项目，应如何操作？
（A）运行File/New命令　（B）运行File/New Project命令
（C）运行Project/New命令　（D）运行Project/New Project...命令

（ ）3. 在Keil μVision 3里，若要将C源文件加入目前的项目应如何操作？
（A）运行File/Add Source File命令
（B）将鼠标指向项目窗口里的Source Group 1项，单击右键，选择Add Files to Group ‘Source Group 1’选项
（C）单击按钮　　（D）按 A 键

（ ）4. 在Keil μVision 3里，若要进行组建，应如何操作？
（A）运行Tools/Build命令（B）单击按钮（C）单击按钮（D）按 C 键

（ ）5. 在Keil μVision 3里，若要打开调试/仿真工具栏，应如何操作？
（A）单击按钮（B）单击按钮（C）单击按钮（D）单击按钮

（ ）6. 在Keil μVision 3里，若要全速进行程序的调试/仿真，应如何操作？
（A）单击按钮（B）单击按钮（C）单击按钮（D）单击按钮

（ ）7. 同上题，若要单步执行程序的调试/仿真，且要能跳过子程序，应如何操作？
（A）单击按钮（B）单击按钮（C）单击按钮（D）单击按钮

（ ）8. 进行调试/仿真时，若想要观察输入/输出端口的状态，可以如何处理？
（A）运行Peripherals/I/O-Ports命令　（B）运行View/Ports命令
（C）运行Edit/Ports命令　　（D）单击按钮

（ ）9. 下列哪项不是Keil C的预处理命令？
（A）#include　（B）#define　（C）#exit　（D）#if

（ ）10. 下列哪项不是Keil C的数据类型？
（A）void　（B）string　（C）char　（D）float

问答题

1. 试述Keil C试用版与商用版最明显的差异。

2. 试述Keil μVision 3环境里，所谓“组建”是进行哪些工作？

3. 若在程序里，所要控制的信号是通过Port 0输出，在Keil μVision 3环境里进行调试时，如何跟踪Port 0的状态？

4. 若在程序里引用8051的中断功能，在Keil μVision 3环境里进行调试时，如何进行中断功能的仿真？

5. 在Keil C程序里，主程序与函数的最明显差异是什么？

6. 在Keil C程序里，若要将“my.h”头文件包含到程序，应如何处理？

7. 试述在说明Keil C程序里如何标注？

8. 试述Keil C提供哪几种存储器形式？哪几种存储器模式？

9. 试说明在Keil C提供哪些基本的数据类型？哪些8051特殊的数据类型？

10. 试说明在Keil C里，“逻辑运算符”与“布尔运算符”有何不同？

11. 试说明在Keil C的“while”与“while-do”语句有何不同？

12. 试述8051内部有多少个寄存器组？如何切换？

13. 试简述PSW是什么？并说明其中各位的功能？

14. 试说明什么是SFR？其地址在哪里？

15. 试简介MCS-51程序的开发流程与工具。

第3章 输出口的应用

本章内容丰富，主要包括以下两部分。

硬件部分：

认识8051的输入/输出端口。

熟悉LED、蜂鸣器、继电器、7段LED数码管的设计技巧。

程序与实践部分：

驱动蜂鸣器实验。

驱动继电器实验。

霹雳灯及7段LED数码管的设计与应用。

3.1 认识8051的输入/输出端口

MCS-51的突出优点之一就是它有4个输入/输出端口！这4个输入/输出端口的结构大致相同，但各有它们的特别之处，说明如下。

PORT 0

PORT 0为8位、可位寻址的输入/输出端口，以双列直插式封装的8051为例，P0.0为39脚，P0.1为38脚……P0.7为32脚，图3-1所示为其中一个位的内部结构。

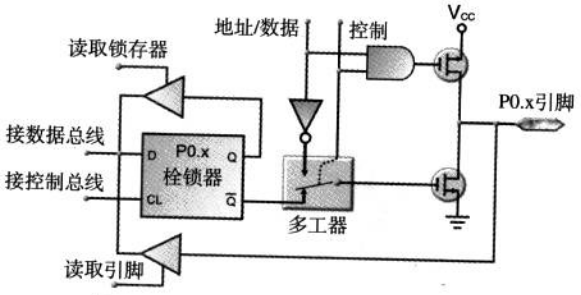


图3-1 PORT 0内部电路结构（1个位）

PORT 0的特点说明如下。

- PORT 0的8位皆为漏极开路输出（Open Drain，简称OD），千万不要误解为图腾式输出，而每个引脚可驱动8个LS型TTL负载。
- PORT 0内部无上拉电阻，所以执行输出功能时，外部必须接上拉电阻（10kΩ即可）。
- 若要实现输入功能，必须先输出高电平（1），才能读取该端口所连接的外部数据。
- 若系统连接外部存储器，则PORT 0可作为地址总线（A0~A7）及数据总线（D0~D7）的多功能引脚，此时内部具有上拉电阻，不用外接上拉电阻。

PORT 1

PORT 1为8位、可位寻址的输入/输出端口，以双列直插式封装的8051为例，P1.0为1脚、P1.1为2脚……P1.7为8脚，如图3-2所示为其中一个位的内部结构：

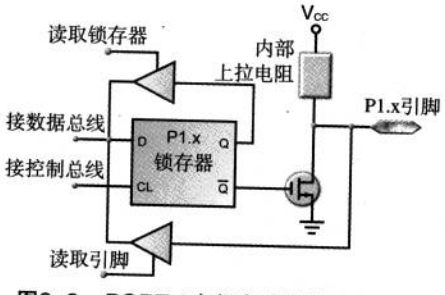


图3-2 PORT 1内部电路结构（1个位）

PORT 1的特色说明如下。

- PORT 1内部具备约30kΩ的上拉电阻，实现输出功能时，不需要连接外部上拉电阻。
- PORT 1的8位类似漏极开路输出（OD），但已经内接了上拉电阻，每个引脚可驱动4个LS型TTL负载。
- 若要执行输入功能，必须先输出高电平（1），才能读取该端口所连接的外部数据。
- 若是8052/8032，则P1.0兼具有Timer 2的外部脉冲输入功能（即T2）、P1.1兼具有Timer 2的捕捉/重新加载的触发输入功能（即T2EX）。
- 若是89S51/89S52，进行在线烧录（ISP）时，其中的P1.5当作MOSI用，P1.6当作MISO用，P1.7当作SCK用。

PORT 2

PORT 2为8位、可位寻址的输入/输出端口，以双列直插式封装的8051为例，P2.0为21脚，P2.1为22脚……P2.7为28脚，如图3-3所示为其中一个位的内部结构。

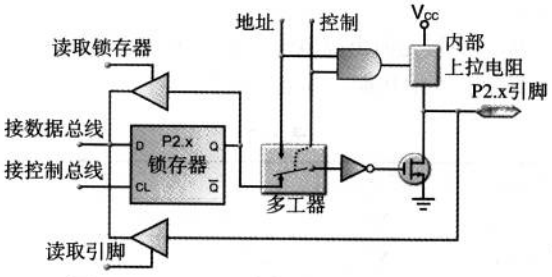


图3-3 PORT 2内部电路结构（1个位）

PORT 2的特色说明如下。

- PORT 2内部具备约30kΩ的上拉电阻，实现输出功能时，不需要连接外部上拉电阻。
- PORT 2的8位类似漏极开路输出（OD），但已经内接了上拉电阻，每个引脚可驱动4个LS型TTL负载。
- 若要实现输入功能，必须先输出高电平（1），才能读取该端口所连接的外部数据。
- 若系统连接外部存储器，而外部存储器的地址线超过8根时，则PORT 2可作为地址总线（A8~A15）引脚。

PORT 3

PORT 3为8位、可位寻址的输入/输出端口，以双列直插式封装的8051为例，P3.0为10脚，P3.1为11脚……P3.7为17脚，如图3-4所示为其中一个位的内部结构。

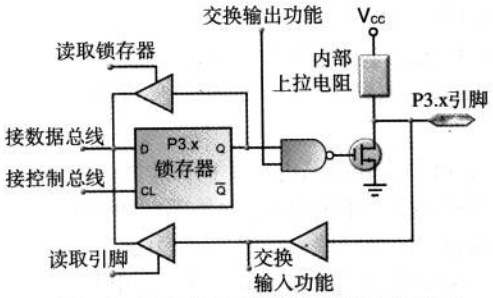


图3-4 PORT 3内部电路结构（1个位）

PORT 3的特色说明如下。

- PORT 3内部具备约30kΩ的上拉电阻，实现输出功能时，不需要连接外部上拉电阻。
- PORT 3的8位类似漏极开路输出（OD），但已经内接了上拉电阻，每个引脚可驱动4个LS型TTL负载。
- 若要实现输入功能，必须先输出高电平（1），才能读取该端口所连接的外部数据。
- PORT 3的8个引脚各有其他功能，如表3-1所示。

表3-1

PORT 3	其他功能	说　明
P3.0	**RXD**	串行口的接收引脚
P3.1	**TXD**	串行口的传送引脚
P3.2	**INT0**	INT0中断输入
P3.3	**INT1**	INT1中断输入
P3.4	**T0**	Timer 0输入
P3.5	**T1**	Timer 1输入
P3.6	**WR**	写入外部存储器控制引脚
P3.7	**RD**	读取外部存储器控制引脚

3.2 输出电路设计



8051的输出端口可直接连接数字电路，也可以用来驱动LED、喇叭或继电器等负载，以下就来探讨如何使用这些负载。

3.2.1 驱动LED

LED为发光二极管（**Light-Emitting Diode**）的简称，其体积小、耗电低，常被用作微型计算机与数字电路的输出装置，用以显示信号状态。近年来LED的技术发展很快，除了能显示红色、绿色、黄色外，还出现了蓝色与白色。亮度很高的LED甚至取代了传统的灯泡，成为交通指示灯（红绿灯）的发光器件，就连汽车的尾灯也开始流行使用LED。

LED具有二极管的特色。逆向偏压时，LED将不发光；顺向偏压时，LED将发光；以红色LED为例，顺向偏压时LED两端约有1.7V左右的压降（比二极管大），如图3-5所示为其特性曲线。

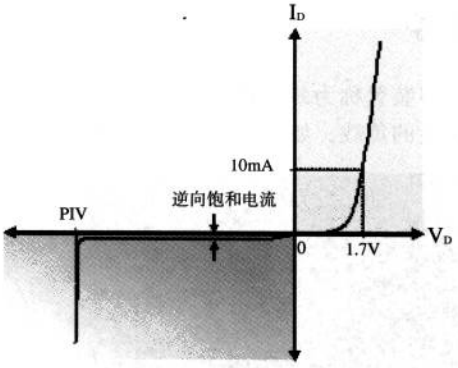


图3-5 LED特性曲线

随着通过LED的顺向电流的增加，LED的亮度将更亮，而LED的寿命也将缩短，所以电流以10~20mA为宜。8051的输入/输出端口都是漏极开路的输出，其中的P1、P2与P3内部有30kΩ的上拉电阻，因此想从P1、P2或P3流出10~20mA，恐怕有困难。如果从外面流入电流到8051的Port，那电流就可以大一点，如图3-6所示。

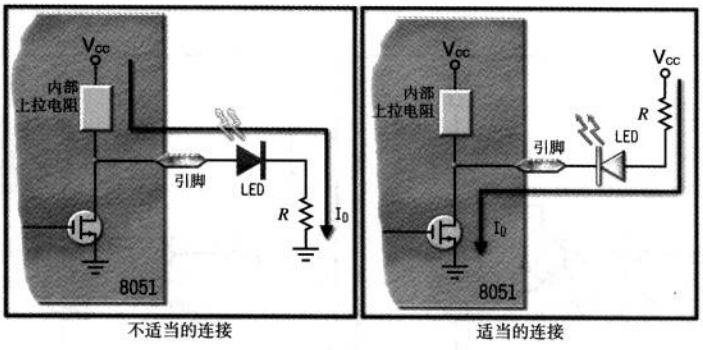


图3-6 输出LED的连接

如图3-6右图所示，当输出低电平时，输出端的FET将导通，输出端电压接近0V，若LED顺向时，两端电压V_D为1.7V，则限流电阻R两端将存在3.3V（即5-1.7）。如果希望流过LED的电流I_D限制为10mA，则此限流电阻R为：

$$R = \frac{5 - 1.7\text{V}}{10\text{mA}} = 330\Omega$$

若想要LED亮一点，可使I_D提高到15 mA，则限流电阻R改为：

$$R = \frac{5 - 1.7\text{V}}{15\text{mA}} = 220\Omega$$

对于TTL电平的数字电路或微型计算机电路，LED所串接的限流电阻大多为

200~330Ω，电阻越小，LED越亮。若LED为非连续负载（例如扫描电路或闪烁灯），则电流还可大一点，甚至可以采用50~100Ω的限流电阻。

3.2.2 驱动蜂鸣器

在微处理电路上的发声装置称为蜂鸣器（buzzer）。一般来说，蜂鸣器就是小型喇叭（speaker），也是一种电感性的负载，如图3-7所示为12mm蜂鸣器的外观与尺寸。

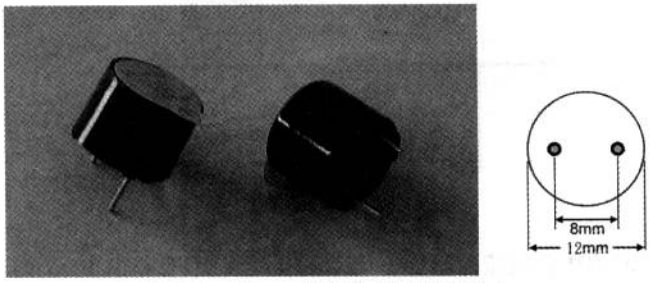


图3-7 12mm蜂鸣器的外观与尺寸

8051驱动蜂鸣器的信号为各种频率的脉冲。因此，最简单的蜂鸣器驱动方式就是利用达林顿晶体管，或以两个常用的小晶体管（如CS9013）连接成达林顿架构，如图3-8所示。

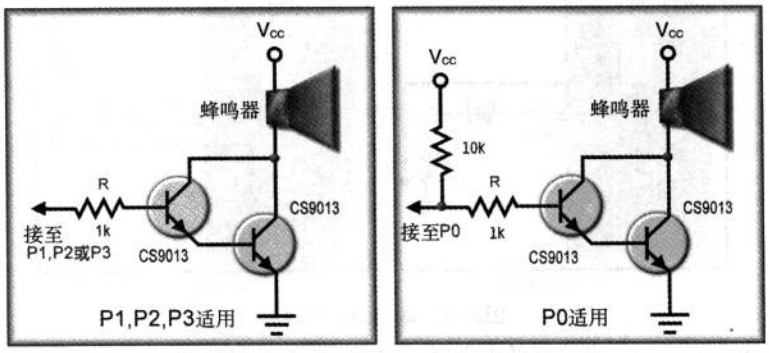


图3-8 使用晶体管驱动蜂鸣器：左图适用于P1、P2、P3，右图适用于P0

其中的电阻R为限流电阻，在此利用晶体管的高电流增益（若单个晶体管的ß超过35，则整体电流增益将超过1 000倍），来达到电路快速饱和的目的。不过，若要由P0口输出连接到此电路，还需连接一个10kΩ的上拉电阻，电路会比较麻烦，如图3-8右图所示。

3.2.3 驱动继电器

如果要利用8051来控制不同电压或较大电流的负载，可通过继电器（RELAY）来传达控制信号。电子电路所使用的继电器，或者说是用在电路板上的继电器，其体积都不大，如图3-9所示为常用的1P继电器。

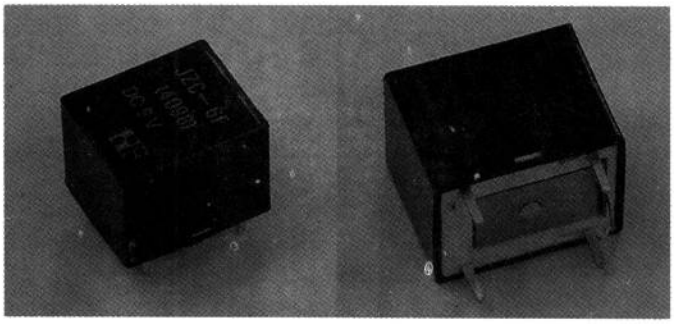


图3-9 电路板上所使用的继电器

这种继电器所使用的电压有DC12V、DC9V、DC6V、DC5V等，通常会直接标示在继电器上面，图3-9所示的继电器就很清楚地标示了电压为DC9V，其引脚图如3-10所示。

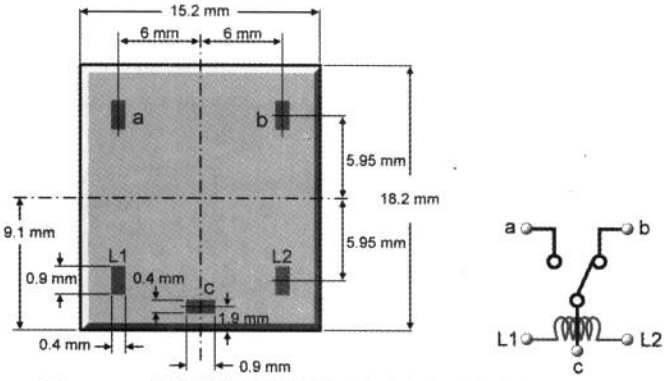


图3-10 继电器的引脚位置图（左）与引脚内部图（右）

其中c-b之间为常闭接点（Normal Closed，简称为NC接点），c-a之间为常开接点（Normal Opened，简称为NO接点），而只有一组a-b-c称为1P。这种继电器很普遍，也不贵，可直接应用在电路板上，但引脚位置并不适用于面包板。

另外，有一种长条形的继电器，其引脚配置可直接使用在面包板上，属于较为精细的电路板用继电器。其上方有个LED，当继电器动作时，上面的LED会亮，如图3-11所示。

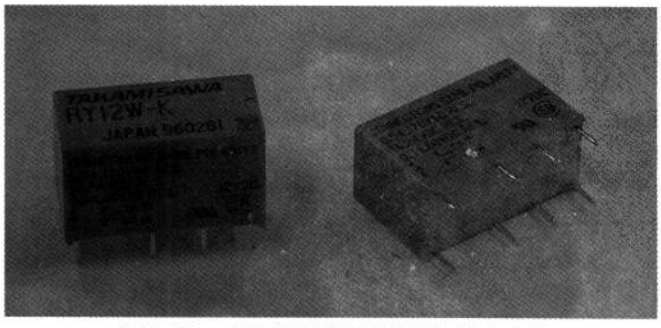


图3-11 电路板上所使用的继电器（2P）

同样地，这种继电器所使用的电压有DC12V、DC9V、DC6V、DC5V等，通常会直接标示在继电器上面，而且引脚图也会印在继电器上，其引脚图如图3-12所示。

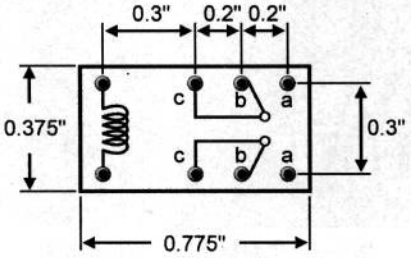


图3-12 继电器的引脚位置图与引脚内部图

通常8051所要驱动的继电器，大多为DC6V或DC5V的小型电子用继电器。尽管如此，光靠8051输出口的电流恐怕不够，况且驱动继电器线圈这种电感性负载，还要有些保护才行！我们可以使用晶体管来控制继电器，以12V继电器为例，如图3-13所示连接电路。

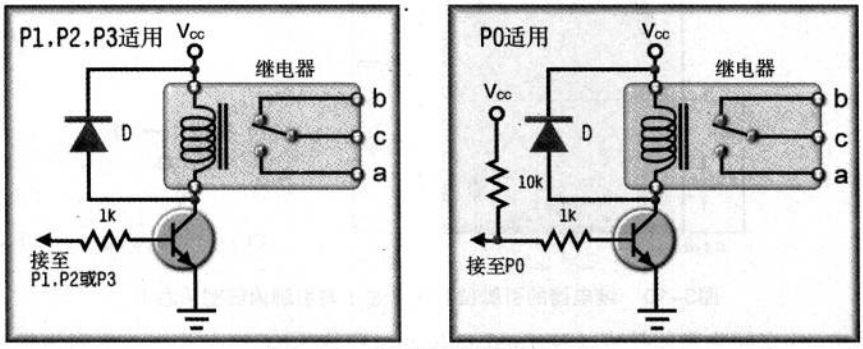


图3-13 使用晶体管驱动继电器

在此，晶体管被当成开关来用。当8051输出高电平时，晶体管工作于饱和状态；当8051输出低电平时，晶体管工作于截止状态。其中的二极管D提供继电器线圈电流的放电路径，以保护晶体管。由于线圈属于电感性负载，当晶体管截止时，i_c=0，而原本线圈上的电流i_L不可能瞬间降为0，所以二极管D就提供一个i_L的放电路径，使线圈不会产生高的应电势，就不会破坏晶体管了。另外，若要由P0输出到此电路，还需要连接一个10kΩ的上拉电阻，如图3-13右图所示。

如果要同时有多个输出端口要驱动继电器，则可使用集电极式（OC）输出的反相闸，如7405（驱动5V的继电器）或7406（驱动较高电压的继电器，最高30V），如图3-14所示。

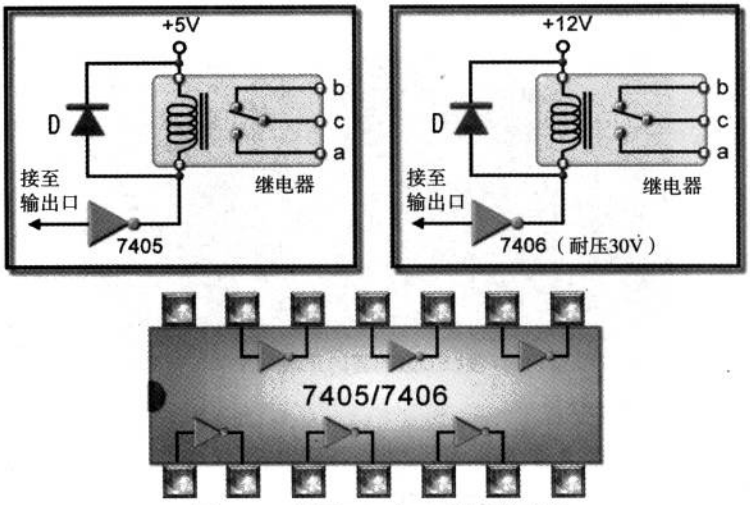


图3-14 使用7405/7406驱动继电器

3.2.4 7段LED数码管

7段LED数码管是利用7个LED组合而成的显示装置，可以显示0到9等10个数字，如图3-15所示。



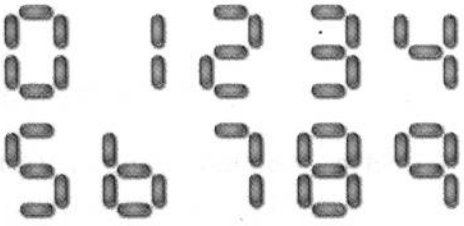


图3-15 7段LED数码管

总地来说，7段LED数码管可分为共阳极与共阴极两种。共阳极就是把所有LED的阳极连接到共同的连接点com，而每个LED的阴极分别为a、b、c、d、e、f、g及dp（**d**ecimal **p**oint，小数点）；同样地，共阴极就是把所有LED的阴极连接到共同的连接点com，而每个LED的阳极分别为a、b、c、d、e、f、g及dp，如图3-16所示。

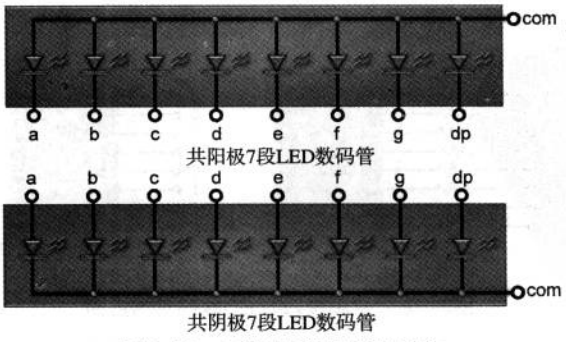


图3-16 7段LED数码管的结构

常用的7段LED数码管，如图3-17所示。

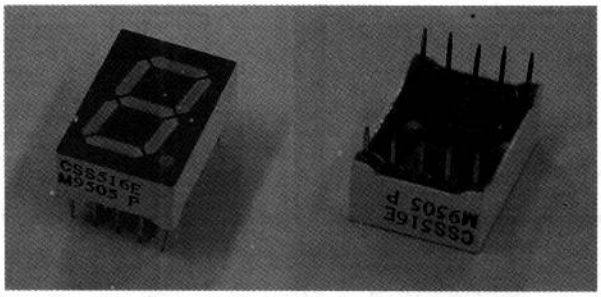


图3-17 常用的7段LED数码管的正面（左）与背面（右）

其引脚与尺寸如图3-18所示。

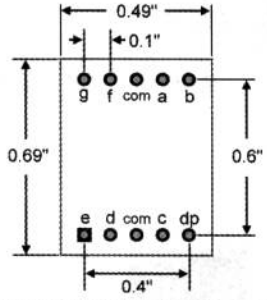
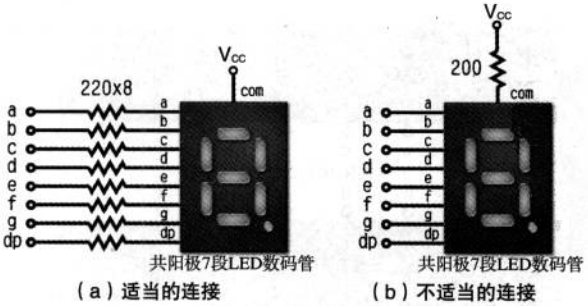


图3-18 常用的7段LED数码管尺寸与引脚配置

共阳极（Common Anode）7段LED数码管

就像一般的LED一样，当我们要使用共阳极7段LED数码管时，首先把com脚接+V_{cc}，然后将每一只阴极引脚各接一个限流电阻，如图3-19（a）所示。在数字或微型计算机电路里，限流电阻可以使用200~330Ω，电阻值越大，亮度越弱、电阻值越小，电流越大，亮度也越大。如图3-19（b）所示，若只使用一个限流电阻，则显示不同数字时将会有不同的亮度，这样就不合适。



（a）适当的连接 （b）不适当的连接

图3-19 共阳极7段LED数码管的应用

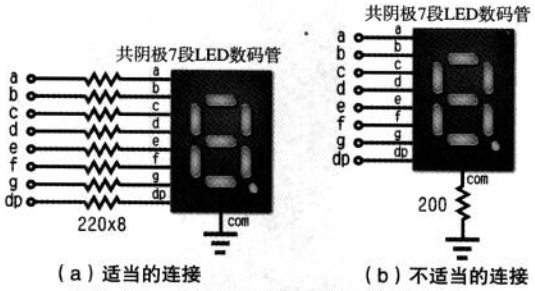
若a连接8051输出端口的最低位（LSB），dp连接8051输出端口的最高位（MSB），且希望小数点不亮，则0至9的驱动信号如表3-2所示。

表3-2

数字	（dp）gfedcba	16进位	显示
0	11000000	0xc0	0
1	11111001	0xf9	1
2	10100100	0xa4	2
3	10110000	0xb0	3
4	10011001	0x99	4
5	10010010	0x92	5
6	10000011	0x83	6
7	11111000	0xf8	7
8	10000000	0x80	8
9	10011000	0x98	9

共阴极（Common Catchode）7段LED数码管

当我们要使用共阴极7段LED数码管时，首先把com脚接地（GND），然后将每一只阳极引脚各接一个限流电阻，如图3-20所示。



（a）适当的连接　　（b）不适当的连接

图3-20　共阴极7段LED数码管的应用

若a连接8051输出端口的最低位（LSB）、dp连接8051输出端口的最高位（MSB），且希望小数点不亮，则0至9的驱动信号如表3-3所示。

表3-3

数字	（dp）gfedcba	16进位	显示
0	00111111	0x3f	0
1	00000110	0x06	1
2	01011011	0x5b	2
3	01001111	0x4f	3
4	01100110	0x66	4
5	01101101	0x6d	5
6	00111100	0x3c	6
7	00000111	0x07	7
8	01111111	0x7f	8
9	01100111	0x37	9

很明显，共阳极7段LED数码管的驱动信号与共阴极7段LED数码管的驱动信号刚好相反，我们只要使用其中一组驱动信号编码即可。万一所使用的编码与7段LED数码管的极性不符，只要在程序里的输出指令中，加一个反相运算符即可。

3.3 实例演练



在本节中将应用前面所介绍的LED、蜂鸣器、继电器、7段LED数码管等，动手验证这些装置的驱动方法。

3.3.1 驱动蜂鸣器实验

● 实验要点

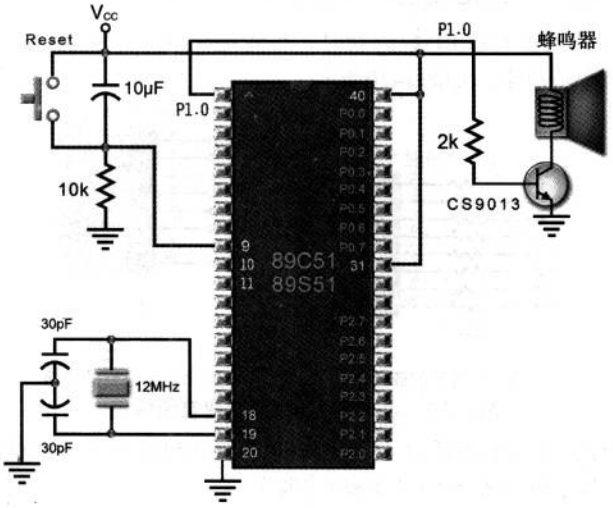


图3-21 蜂鸣器实验电路

1. 如图3-21所示，由P1.0经过晶体管驱动蜂鸣器。

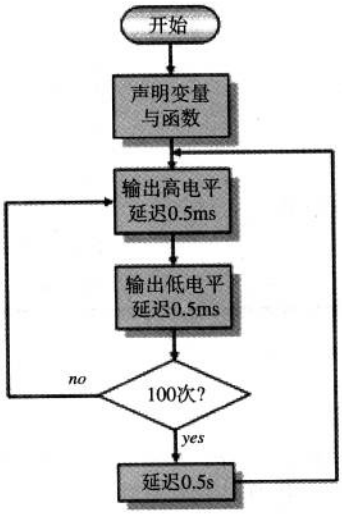
2. 声音是由蜂鸣器的振动产生的。蜂鸣器就像是一个电磁铁，电流流过它即可产生磁性，这样蜂鸣器里发声的膜片将被吸住；电流消失时，膜片将被放开。若要产生频率为f的脉冲，则需要在T时间内（其中$T=\frac{1}{f}$），进行吸、放各一次，换言之，产生磁性、消除磁性的时间各为$\frac{T}{2}$，称为半周期。例如要产生1kHz的频率，则半周期为0.5ms，所以P1.0所送出的信号的一个周期中，其中0.5ms为高电平，另外0.5ms为低电平。若0.5ms为高电平与0.5ms为低电平为一组信号（总共1ms），连续送出100组，即可得到1kHz的声音约0.1s；停止输出0.1s后，再连续送出100组0.5ms为高电平与0.5ms为低电平信号，则可听到“哔哔”两声。

3. 若要P1.0输出高电平，则可利用“P1^0=1;”指令；同理，若要P1.0输出低电平，则可利用“P1^0=0;”指令。另外，可利用delay函数产生0.5ms的延迟。

4. 再使用for（i=0;i<100;i++）的循环语句，即可产生100组驱动信号。

5. 本实验将产生1kHz信号持续0.1s，停0.1s，再产生1kHz信号持续0.1s，停0.5s，然后再从头执行。

流程图与程序设计




```
/* ch03_1.c - 蜂鸣器实验程序 */
//==声明区=====================================
#include        <reg51.h>                  // 定义8051寄存器的头文件，P2-17~19
sbit buzzer = P1^0 ;                       // 声明蜂鸣器的位置为Port1的bit0
void delay(int);                           // 声明延迟函数
void pulse_BZ(int, int, int);              // 声明蜂鸣器发声函数
//==主程序=====================================
main()                                     // 主程序开始
{       while(1)                           // 无穷循环，程序一直运行
        {       pulse_BZ(100，1，1);
                                           // 蜂鸣器发声100×(0.5m+0.5m)=0.1s
                delay(1000);               // 延迟1000×0.5m=0.5s
        }                                  // while循环结束
}                                          // 主程序结束
//==子程序=====================================
/* 延迟函数开始，延迟x×0.5ms */
```

```
void delay(int x)                                    // 延迟函数开始
{       int i, j;                                    // 声明整数变量i，j
        for (i=0;i<x;i++)                            // 计数x次，延迟约x×0.5ms
                for (j=1;j<=60;j++);                 // 计数60次，延迟约0.5ms
}                                                    // 延迟函数结束
/* 蜂鸣器发声函数，count=计数次数，TH=高电平时间，TL=低电平时间 */
void pulse_BZ(int count, int TH, int TL)             // 蜂鸣器发声函数开始
{       int i;                                       // 声明整数变量i
        for(i=0;i<count;i++)                         // 计数count次
        {       buzzer=1;                            // 输出高电平
                delay(TH);                           // 延迟TH×0.5ms
                buzzer=0;                            // 输出低电平
                delay(TL);                           // 延迟TL×0.5ms
        }                                            // for循环结束
}                                                    // 蜂鸣器发声函数结束
```

ch03-1.c

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图3-21连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 若希望产生1kHz声音0.2s、暂停0.05s、600声音0.1s、暂停0.2s，应如何修改？

2. 请调整输出信号的频率与周期，产生电话铃声。

3.3.2 驱动继电器实验

实验要点

1. 如图3-22所示，由P1.0经过晶体管驱动蜂鸣器，P1.1经过晶体管驱动继电器。

2. 总地来说，继电器的工作原理与蜂鸣器的工作原理（详见3.3.1节）类似。当我们

要继电器产生磁性即激磁时，即需要P1.1输出高电平，则可利用“P1^1=1；”指令；若要P1.1输出低电平，则可利用“P1^1=0；”指令。而继电器与蜂鸣器不同之处在于蜂鸣器的操作频率较高，而继电器的操作频率较低。另外，继电器可控制外接电路，例如外接AC110V的电灯泡等负载。当然，一定要看清楚继电器上所标示的接点容量，以图3-11的继电器为例，其上面标示：

0.5A, 120VAC RES.

1A, 24VDC RES.

0.3A 60VDC RES.

表示该接点可控制0.5A/120VAC电阻性负载，或1A/24VDC电阻性负载，或0.3A/60VDC电阻性负载。以0.5A/120VAC电阻性负载为例，即一般插座（AC110V），而0.5A负载电流可驱动55W电阻性负载，所以这个小小的继电器可用来控制40W或50W电灯泡是没有问题的！

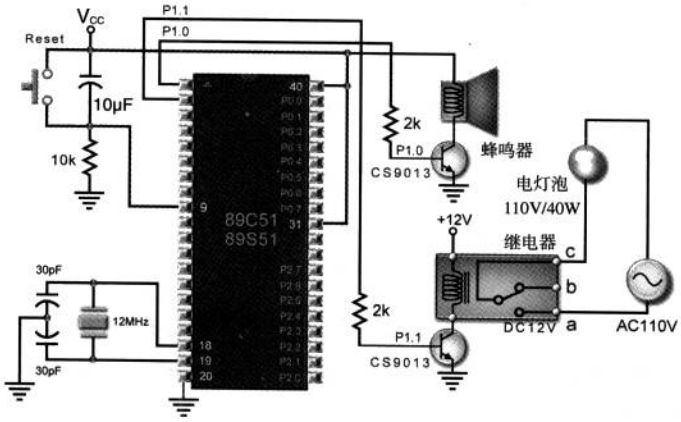
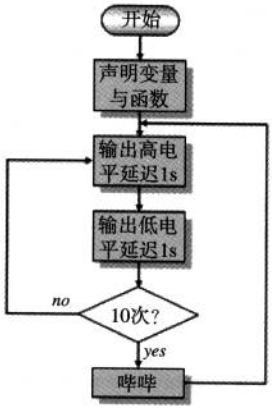


图3-22 继电器实验电路

3. 在本实验的目的是让P1.1所连接的继电器，每1s开关一次，而开关10次后，蜂鸣器“哔哔”两声，然后从头开始执行。

● 流程图与程序设计



```
/* ch03_2.c - 继电器实验程序 */
//==声明区=========================================
#include      <reg51.h> // 定义8051寄存器的头文件，P2-17~19
sbit     buzzer = P1^0;                    // 声明蜂鸣器的位置
sbit     relay = P1^1;                     // 声明继电器的位置
void delay(int);                           // 声明延迟函数
void pulse_BZ(int, int, int);              // 声明蜂鸣器发声函数
void pulse_RL(int, int, int);              // 声明继电器控制函数
//==主程序=========================================
main()                                     // 主程序开始
{       while(1)                           // 无穷循环，程序一直运行 */
        {       pulse_RL(10, 2000, 2000);
                // 继电器使灯亮灭各10次，各2000×0.5m=1s
                pulse_BZ(100, 1, 1);
                // 蜂鸣器第1声哔，约100*(0.5m+0.5m)=0.1s
                delay(200);                // 延迟200×0.5ms=0.1s
                pulse_BZ(100，1，1);
                // 蜂鸣器第2声哔，约100*(0.5m+0.5m)=0.1s
                delay(200);                // 延迟200×0.5ms=0.1s
        }                                  // while循环结束
}                                          // 主程序结束
//==子程序=========================================
// 延迟函数开始，延迟约x*0.5ms/
void delay(int x)                          // 延迟函数开始
{       int i，j;                          // 声明整数变量i，j
        for(i=0;i<x;i++)                   // 计数x次，延迟约x×0.5ms
                for(j=1;j<=60;j++);        // 计数60次，延迟约0.5ms
}                                          // 延迟函数结束
/* 蜂鸣器发声函数，count=计数次数，TH=高电平时间，TL=低电平时间 */
void pulse_BZ(int count, int TH, int TL)   // 蜂鸣器发声函数开始
{       int i;                             // 声明整数变量i/
        for(i=0;i<count;i++)               // 计数count次
        {       buzzer=1;                  // 输出高电平
                delay(TH);                 // 延迟TH×0.5ms
                buzzer=0;                  // 输出低电平
                delay(TL);                 // 延迟TL×0.5ms
        }                                  // for循环结束
}                                          // 蜂鸣器发声函数结束
```

```
/* 继电器控制函数，count=计数次数，TH=激磁时间，TL=消磁时间 */
void pulse_RL(int count, int TH, int TL)  // 继电器控制函数开始
{       int i;                            // 声明整数变量i
        for(i=0;i<count;i++)              // 计数count次
        {       relay=1;                  // 输出高电平，继电器激磁(灯亮)
                delay(TH);                // 延迟TH×0.5ms
                relay=0;                  // 输出低电平，继电器消磁(灯灭)
                delay(TL);                // 延迟TL×0.5ms
        }                                 // for循环结束
}                                         // 继电器控制函数结束
```

ch03-2.c

● 操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图3-22连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

● 思考一下

若要让继电器激磁10s，消磁5s，如此循环，应如何修改？

3.3.3 霹雳灯实例演练

● 实验要点

1. 如图3-23所示，由P2驱动8个LED。当输出0时，LED亮、输出1时，LED不亮。

2. 所谓“霹雳灯”是指在一排LED里（在此使用8个），任何一个时间只有单一个LED亮，而亮灯的顺利为由左而右，再由右而左，感觉上就像一个LED由左跑到右，再由右跑到左，如图3-24所示。

3. 在程序设计上，有很多方法可以达到这个目的。例如采用计次循环方式，如图3-24所示，首先左移7次，再右移7次，如此循环不停。左移可采用“LED<<=1；”指令，右移可采用“LED>>=1；”指令。而计次循环方式，使用for语句即可达到目的。

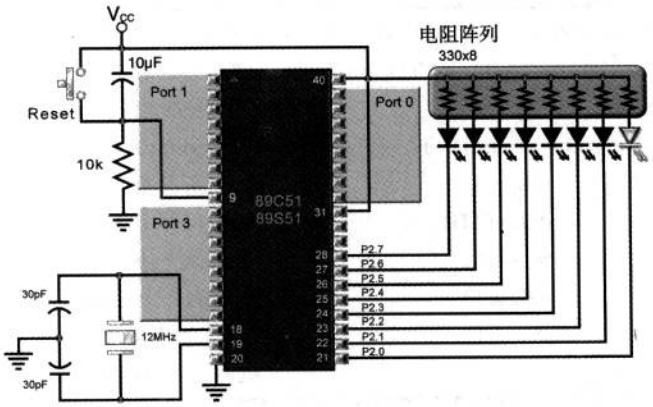


图3-23 霹雳灯实验电路

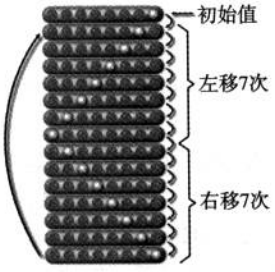
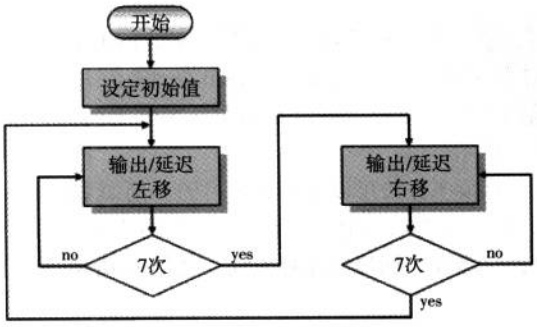


图3-24 霹雳灯动作示意图

4. LED的初始值为11111110（1不亮、0亮），左移时，右边将移入0，变成11111100，所以必须将最右边位改为1。我们可在左移后再利用OR运算，即“**LED=（LED<<=1）|0x01；**”指令，即可将11111110变成11111101。同理，在进行右移时，可应用“**LED=（LED>>=1）|0x80；**”指令。

5. 另外，也可以采用判断的方式实现。以左移为例，原本为LED的内容为“11111110”，左移7次后变成为“01111111”。因此，发现LED的内容为“10000000”时，表示要改变为右移。同样地，右移7次后又变成为“00000001”，这又是进行左移的判断条件。如此循环执行，即可实现霹雳灯效果。

流程图与程序设计



```
/* ch03_3.c - 霹雳灯实验程序 */
//==声明区=============================================
#include   <reg51.h>                    // 定义8051寄存器的头文件，P2-17~19
#define LED    P2                       // 定义LED接到Port 2
void delay(int);                        // 声明延迟函数
//==主程序=============================================
main()                                  // 主程序开始
{       unsigned char i;                // 声明无号数字元变量i(占1B)
        LED=0xfe;                       // 初值=1111 1110，只有最右1灯亮
        while(1)                        // 无穷循环，程序一直运行
        {       for(i=0;i<7;i++)        // 左移7次
                {  delay(100);          // 延迟100×5m=0.5s
                   LED=(LED<<1)|0x01; // 左移1位，并设定最低位为1
                }                       // 左移结束，只有最左1灯亮
                for(i=0;i<7;i++)        // 右移7次
                {  delay(100);          // 延迟100×5m=0.5s
                   LED=(LED>>1)|0x80; // 右移1位，并设定最高位为1
                }                       // 结束右移，只有最右1灯亮
        }                               // while循环结束
}                                       // 主程序结束
//==子程序=============================================
/* 延迟函数，延迟约x×5ms */
void delay(int x)                       // 延迟函数开始
{       int i，j;                       // 声明整数变量i，j
        for (i=0;i<x;i++)// 计数x次，延迟x×5ms
                for (j=1;j<=600;j++);   // 计数600次，延迟5ms
}                                       // 延迟函数结束
```

ch03-3.c

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图3-23连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 了解了单灯的霹雳灯程序之后，请修改该程序，让它变成双灯的霹雳灯功能。

2. 请以“实验要点”第4点所提出的方法，重新编写单灯的霹雳灯程序，并验证其功能。

3.3.4 驱动7段LED数码管实验

实验要点

1. 如图3-25所示，由P0驱动7段LED数码管，其中使用220Ω电阻作为限流电阻。

2. 7段LED数码管上所显示的数字，从0开始，每隔0.5s增加1，直到9之后，再从0开始，如此循环不停。

3. 在本实验的程序里，将利用数组方式储存驱动信号的编码，再逐一将数组里的数据输出到Port 2即可。

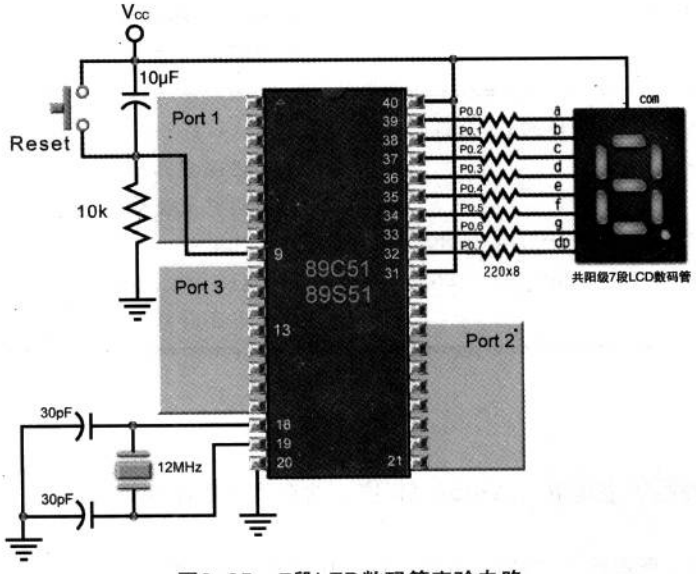
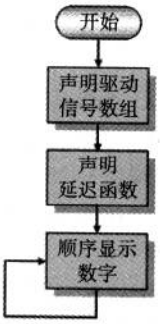


图3-25 7段LED数码管实验电路

流程图与程序设计




```
/* ch03_4.c - 7段LED数码管实验程序 */
//==声明区=====================================
#include<reg51.h> // 定义8051寄存器的头文件，P2-17~19
#define SEG    P0                    // 定义7段LED数码管接到Port 0
/* 声明7段LED数码管驱动信号数组(共阳) */
char TAB[10]={ 0xc0,  0xf9,  0xa4,  0xb0,  0x99,     //数字0~4
        0x92,  0x83,  0xf8,  0x80,  0x98 };          //数字5~9
void delay(int );                    // 声明延迟函数
//==主程序=====================================
main()                               // 主程序开始
{       unsigned char i; // 声明无号数字元变量i
        while(1)                     // 无穷循环，程序一直运行
          for(i=0;i<10;i++)          // 显示0~9，共10次
          {   SEG=TAB[i];            // 显示数字
              delay(500);            // 延迟500 × 1m=0.5s
          }                          // for循环结束
}                                    // 主程序结束
//==子程序=====================================
/* 延迟函数，延迟约x × 1ms */
void delay(int x)                    // 延迟函数开始
{   int i, j;                        // 声明整数变量i, j
    for (i=0;i<x;i++)                // 计数x次，延迟x × 1ms
        for (j=1;j<=150;j++);        // 计数150次，延迟1ms
}                                    // 延迟函数结束
```

ch03-4.c

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图3-25连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 请修改本实验里的程序，让7段LED数码管从9开始显示，递减到0，再从头开始？

2. 请修改本实验里的程序，让7段LED数码管从0开始显示，递增到9，再递减到0，然后从头开始？

3.4 实时练习



在本章里，介绍了8051的输入/输出端口、输出电路的设计等硬件部分；在软件方面，更进行四项外围装置的控制实验。在此请试着回答下列问题，以验证学习成效。

选择题

（ ）1. 在8051的输入/输出端口里，哪个输入/输出端口执行在输出功能时没有内部有上拉电阻？
（A）P0 （B）P1 （C）P2 （D）P3

（ ）2. 在Keil C的程序里，若要指定Port 0的bit 3，如何编写程序？
（A）P0.3 （B）Port 0.3 （C）P0^3 （D）Prot 0^3

（ ）3. 8051的输入/输出端口都是采用哪种电路结构？
（A）集电极式输出 （B）漏极开路输出 （C）射极输出 （D）图腾式输出

（ ）4. 在8051里，若要扩展外部存储器，数据总线连接哪个输入/输出端口？
（A）P0 （B）P1 （C）P2 （D）P3

（ ）5. 点亮一般的LED所耗用的电流约为多少？
（A）1~20μA （B）10~20μA （C）1~20mA （D）10~20mA

（ ）6. 一般蜂鸣器属于哪种负载？
（A）电阻性负载 （B）电感性负载 （C）电容性负载 （D）不导电负载

() 7. 7405/7406的输出采用哪种电路结构?

(A) 集电极式输出 (B) 漏极开路输出 (C) 射极输出 (D) 图腾式输出

() 8. 在继电器里，所谓NO接点是一种什么接点?

(A) 不使用的接点 (B) 不存在的接点

(C) 激磁后即开路的接点 (D) 常开接点

() 9. 所谓2P的继电器代表什么意思?

(A) 只有2个接点 (B) 两相的负载 (C) 两组电源 (D) 2组c接点

() 10. 共阳极7段LED数码管的驱动信号有何特点?

(A) 低电平点亮 (B) 低电平不亮 (C) 高电平点亮 (D) 以上皆非

简答题

1. 试述P0与P2引脚的其他功能。

2. 试述P3引脚的其他功能。

3. 试述7405与7406的异同。

4. 在晶体管驱动继电器的电路里，继电器的线圈两端并接一个逆向二极管，其功能是什么。

5. 试编写一个约1s的延迟函数。

() 7. 74LS273和74LS244输出类型都是什么?

(A) 集电极开路输出 (B) 三态门输出 (C) 推挽输出 (D) 图腾式输出

() 8. 继续电路里，所谓的锁存是什么意思?

(A) 不使用的状态 (B) 三态门的状态

(C) 断路后的开路的状态 (D) 高电平状态

() 9. 74LS245的缓冲器是什么意思?

(A) 只有2个按点 (B) 两组的输出 (C) 输出电源 (D) 3脚按点

() 10. 使用按压式开关的缺点是什么意思?

(A) 机电寿命短 (B) 维护麻烦 (C) 需电子点亮 (D) 以上皆非

实作

1. 写出下列各74LS273的其功能。

2. 按图示电路的其功能。

3. 写出74LS273与74LS245各同。

4. 在指定器的输出器的电路里，要自动式图所需的一个字向二位数，其功能

第4章 输入口的应用



本章内容丰富，主要包括以下两部分。

硬件部分：

介绍8051的输入口。

介绍常用的按钮开关、指拨开关、数字型指拨开关等及其应用。

程序与实践部分：

指拨开关的应用。

按钮开关的应用。

数字型指拨开关的应用。

4.1 认识输入口

在3.1节里，我们绘出8051 4个输入/输出端口的结构（图3-1~图3-4）。虽然这4个输入/输出端口的结构有些不同，但就输入功能来看，这4个输入/输出端口的结构几乎完全一样！基本上，输入口都是通过一个三态的寄存器连接到CPU内部的数据总线，以PORT 0为例，它的结构如图4-1所示。

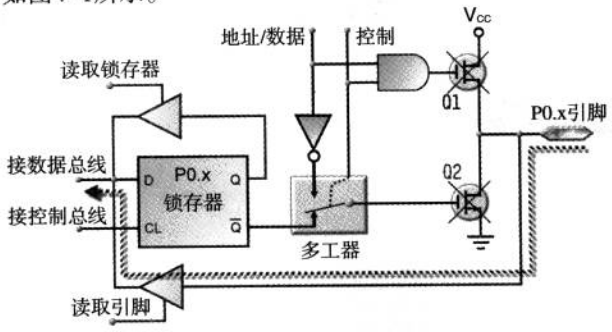


图4-1 PORT 0的输入功能

在实现输入功能时，输出端的Q1、Q2两个FET必须呈开路状态才不会影响输入状态。而进行一般数据的输入/输出时，Q1就是高阻抗状态（可看作开路）。若要Q2也呈高阻抗状态，其闸极必须为低电平，而其闸极连接多任务器，再连接到锁存器的 $\overline{Q}$；若要让锁存器的 $\overline{Q}$ 为低电平，则其输入端D必须为高电平。换言之，只要该位输出1，则内部数据总线该位为1，锁存器的输入端D为1，其输出Q=1、$\overline{Q}$=0，并由Q回传到输入端，使该锁存器保持Q状态；而当 $\overline{Q}$=0时，Q2将呈高阻抗状态。这也就是为什么在输入之前，必须送“1”到该输入/输出端口，将该输入/输出端口设计成输入功能的原因。若没有事先将“1”到该输入/输出端口，则Q2可能不是高阻抗，可能会影响输入的状态。这就是为什么没有事先将“1”送到该输入/输出端口，还是可以把该输入/输出端口当成输入口的原因。不过，最好还是规规矩矩地先输出“1”比较可靠。

当要输入该位引脚所连接的外部数据时，输入指令将使内部“读取引脚”线变为1，外部数据才会通过寄存器，送到内部数据总线。

4.2 输入设备与输入电路设计

在此所要介绍的是与人们接触较为频繁的输入设备，包括电子电路常用的按钮开关、指拨开关等。

4.2.1 输入设备

对于数字电子电路而言，最基本的输入设备就是开关。开关可以分为按钮开关和闸刀开关两类，说明如下。

- 按钮开关（Button）的特色就是具有自动恢复（弹回）的功能。我们按下按钮，其中的接点接通（或切断），放开按钮后，接点恢复为切断（或接通）。在电子电路方面，最典型的按钮开关就是小小的Tack Switch，如图4-2所示。当然，在工业应用中也会以导电橡皮所组成的按钮来降低成本，尤其是同时需要多个按钮的键盘组。

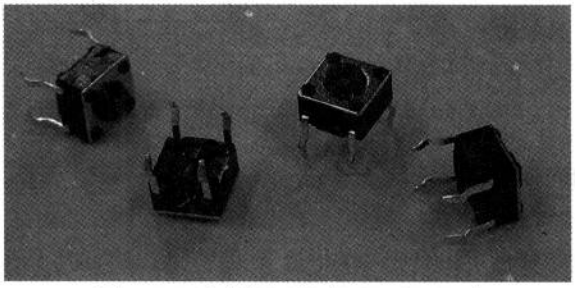


图4-2 4个按钮开关（Tack Switch）

- 闸刀开关（Knife Switch）具有保持功能，也就是不会自动复归（弹回）。当我们按一下开关（或切换开关）时，其中的接点接通（或切断），若要恢复接点状态，则需再按一下开关（或切换开关）。在电子电路方面，最典型的闸刀开关就是指拨开关（DIP switch），如图4-3所示。当然，对于电路板的状态设置不常切换开关状态的场合，也常用跳线（Jumper）来代替，也就是在电路板上放置两个引脚的排针，然后用跳线环（短路环）作为接通的部件。

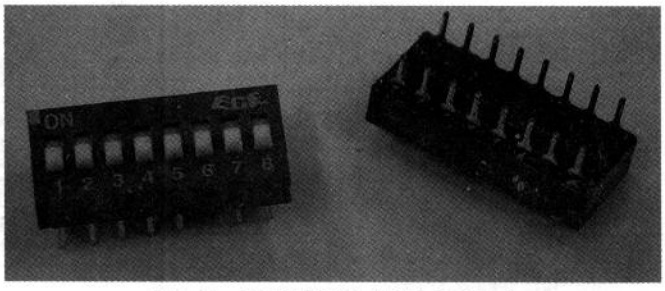


图4-3 8P指拨开关（DIP Switch）

以下将介绍这两类开关的相关器件。

按钮开关（Tack Switch）

按照尺寸区分，电子电路或微型计算机所使用的Tack Switch可分为8mm、10mm、12mm等。虽然Tack Switch有4个引脚，但实际上，其内部只有一对a接点。如图4-4所示，在尺寸图之中，上面两个引脚是内部相连通的，而下面两个引脚也是内部相连通的。上、下之间则为一对a接点。

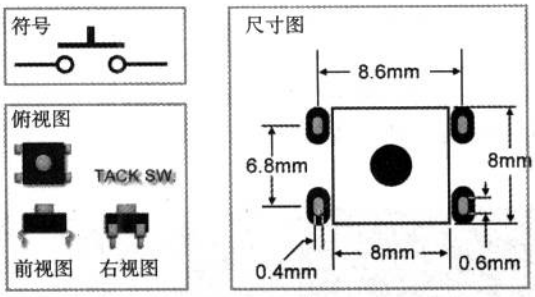


图4-4　8 mm Tack Switch的符号、外观与尺寸

指拨开关（DIP Switch）

按指拨开关的开关数量，可分为2P、4P、8P等。2P指拨开关内部有独立的两个开关，4P指拨开关内部有独立的4个开关，8P指拨开关内部有独立的8个开关。通常会在DIP Switch上标示记号或“ON”，若将开关拨到记号或“ON”的一边，则接点接通（on）、拨到另一边则为不通（off）。如下图所示为8P指拨开关的符号、三视图与尺寸图。

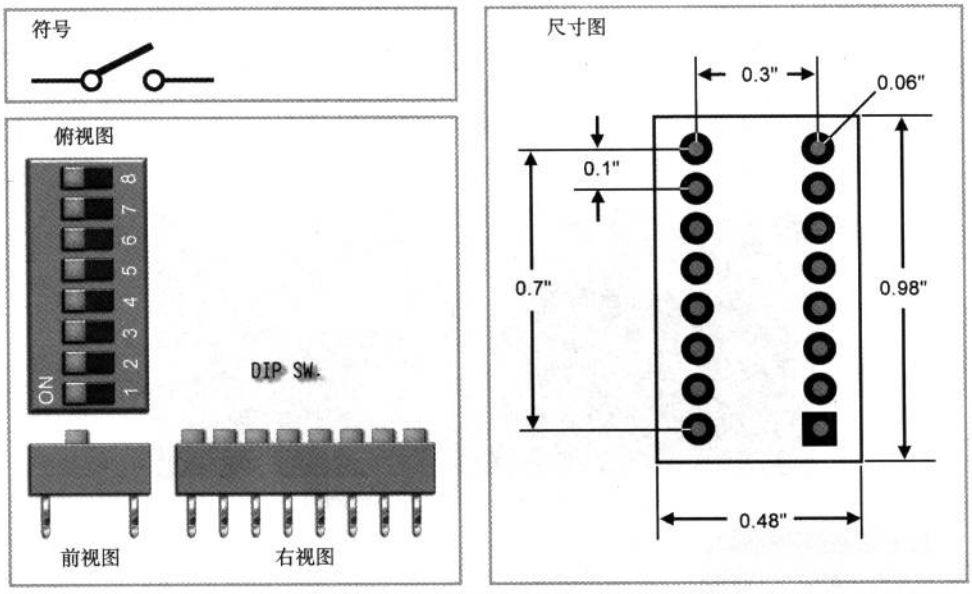


图4-5　8P DIP Switch的符号、外观与尺寸

数字型指拨开关

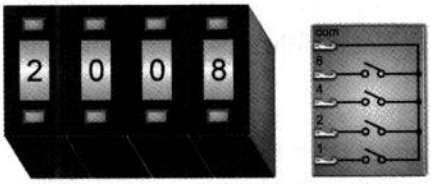


图4-6　数字型指拨开关的外观（四位数）与内部结构（每一位数）

数字型指拨开关是一种附有数字轮盘的指拨开关。如图4-6所示。根据其数字编码区分，数字型指拨开关可分为下列两种类型。

- **BCD**指拨开关提供0~9的BCD编码输出。如表4-1所示为开关输出状态，而其数字轮盘只有10个（0~9）。
- **16**进制指拨开关提供0~F的16进制编码输出。如表4-1所示为开关输出状况，而其数字轮盘有16个（0~F）。

表4-1

类型		数字	8输出端	4输出端	2输出端	1输出端
16进制	BCD	0	OFF	OFF	OFF	OFF
		1	OFF	OFF	OFF	ON
		2	OFF	OFF	ON	OFF
		3	OFF	OFF	ON	ON
		4	OFF	ON	OFF	OFF
		5	OFF	ON	OFF	ON
		6	OFF	ON	ON	OFF
		7	OFF	ON	ON	ON
		8	ON	OFF	OFF	OFF
		9	ON	OFF	OFF	ON
		A	ON	OFF	ON	OFF
		B	ON	OFF	ON	ON
		C	ON	ON	OFF	OFF
		D	ON	ON	OFF	ON
		E	ON	ON	ON	OFF
		F	ON	ON	ON	ON

根据其切换方式区分，数字型指拨开关可分为下列两种类型：

- 上下按钮式切换，在数字上方有个按钮，按此按钮，数字将减1；在数字下方也有个按钮，按此按钮，数字将加1。
- 旁边转盘式切换，在数字旁边有个轮盘，直接旋转轮盘，即可操作显示的数字。

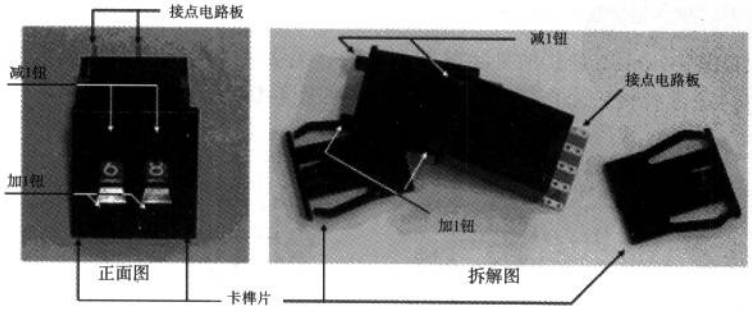


图4-7 数字型指拨开关的实体图

另外，数字型指拨开关属于位数独立的设备，需要几位数，就购买几片数字型指拨开关，再把它们组合起来即可。例如要6位数，则购买6片，再把并排、对准卡榫，压入即可。而且直接拉开即可拆开。最后，左右两旁各压入一片卡榫（数字型指拨开关所附的），即可将它嵌入机壳。一般来说，数字型指拨开关是直接装在机壳上让使用者操作的，再通过排线将其接点连接到电路板上。

4.2.2 输入电路设计

当我们要设计数字电路或微处理电路的输入电路时，一定要把握一个原则，就是不要有不确定的状态。所以，输入端不可留空，输入端留空除了会产生不确定的状态外，还可能感染噪声，使电路产生错误的操作。以下针对按钮开关与闸刀开关，说明其相应输入电路的设计。

● 按钮开关的输入电路设计

不管是Tack Switch还是其他种类的按钮开关，若要将它作为数字电路或微型计算机电路的输入时，通常会接一个电阻到V_{cc}或GND，如图4-8所示。

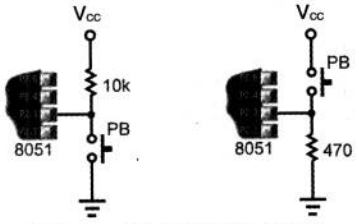


图4-8 按钮开关的输入电路

如图4-8左图所示，平时按钮开关（PB）为开路状态，其中10kΩ的电阻连接到V_{cc}，使输入引脚上保持为高电平信号；若按下按钮开关，则经过开关接地，输入引脚上将变为低电平信号；放开开关时，输入引脚上将恢复为高电平信号，这样将可产生一个负脉冲。反之，如图4-8右图所示，平时按钮开关为开路状态，其中470Ω的电阻接地，使输入引脚上保持为低电平信号；若按下按钮开关，则经过开关接V_{cc}，输入引脚上将变为高电平信号；放开开关时，输入引脚上将恢复为低电平信号，这样将可产生一个正脉冲。

● 闸刀开关的输入电路设计

不管是DIP Switch还是其他种类的闸刀开关，若要将它作为数字电路或微型计算机电路的输入时，通常会接一个电阻到V_{cc}或GND，如图4-9所示。

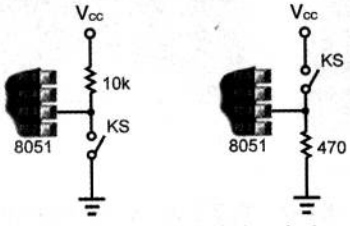


图4-9 闸刀开关的输入电路

如图4-9左图所示，若开关（KS）为off状态，其中10kΩ的电阻连接到V_{cc}，使输入引脚上保持为高电平信号；若将开关切换到on的状态，则经过开关接地，输入引脚上将变为低电平信号，这样将可根据需要产生不同的电平。反之，如图4-9右图所示，若开关为off状态，其中470Ω的电阻接地，使输入引脚上保持为低电平信号；若将开关切换到on的状态，则经过开关接V_{cc}，输入引脚上将变为高电平信号，如此将可以根据需要产生不同的电平。

> 通常按钮开关是使用在产生边缘触发的场合，而闸刀开关使用在产生电平触发的场合。

数字型指拨开关

每片数字型指拨开关都有5个接点，分别是com、8、4、2、1，通常是把com连接V_{cc}，而其他端点分别通过一个470Ω的电阻接地，如图4-10所示。

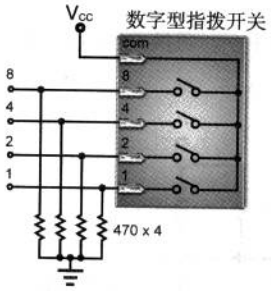


图4-10 数字型指拨开关的应用电路

若要把数字型指拨开关与89S51连接，则把图4-10中所示的8、4、2、1端直接并接于输入口即可，其中8端是MSB、1端是LSB，以连接P2为例，其电路如图4-11所示。

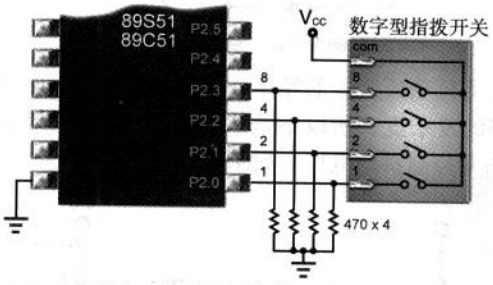


图4-11 BCD指拨开关的使用

4.2.3 抖动与防抖动

不管是按钮开关还是闸刀开关，在操作时，并不如想象中那么理想！实际上，开关操作时会有很多不确定状况，也就是噪声。在此将介绍开关操作的实际状况以及防止不确定状况的对策。

抖动现象

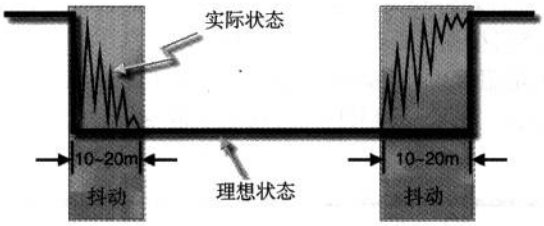


图4-12 开关的操作

在刚才所介绍的输入电路中，开关的操作是理想的状态，如图4-12中浅蓝色线所示的波形。但如果仔细分析开关的真实操作，将可发现许多非预期的状况，如图4-12中的锯齿状线所示的波形。这种非预期的状况称为抖动（bouncer），而这种忽高忽低，忽而非高非低，可说是不折不扣的噪声。

硬件防抖动

如果要避免这种现象，可使用一个切换开关（c接点）及互锁电路，组成一个防抖动电路（debouncer），如图4-13所示。

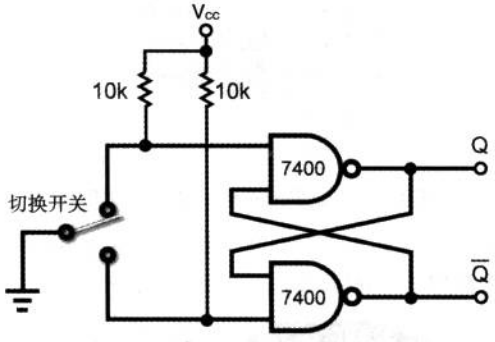


图4-13 互锁电路

虽然这个电路可降低抖动所产生的噪声，但所需要的零件较多，所占的电路面积较大，增加了成本与电路的复杂度，所以不是非不得已，已经很少使用了。当然，我们可利用一个简单的RC电路来控制抖动电压，如图4-14所示。

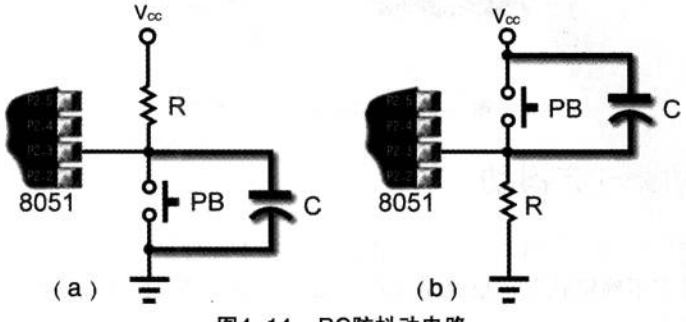


图4-14 RC防抖动电路

以图4-14（a）为例，当按下按钮开关时，开关第一次接触时，即将电容短路，使电容快速放电（放电电阻为0），电容两端电压迅速降为0；开关弹回（开路）时，整个电路形成RC充电电路，其时间常数为RC，电容两端的电压V_C为

$$V_C = V_{CC} \times (1 - e^{-\frac{t}{RC}})$$

通常低电平可以定义为 $0.3 \times V_{CC}$ 以下，如果电容两端的电压V_C低于 $0.3 \times V_{CC}$，即可看作低电平，而抖动的效应自然消失。因此

$$V_C = VCC \times (1 - e^{-\frac{t}{RC}}) < 0.3 \times V_{CC}$$

即$1 - e^{-\frac{t}{RC}} < 0.3$，两边减1可得$-e^{-\frac{t}{RC}} < -0.7$，再把两边改号，小于变大于，即

$$e^{-\frac{t}{RC}} > 0.7$$

两边取对数

$$-\frac{t}{RC} > \ln 0.7 \cong -0.357$$

两边再改号，即

$$\frac{t}{RC} < 0.357$$

抖动的时间约在10ms到20ms之间，以10ms为例，若R=10kΩ，则

$$\frac{10m}{10k \times C} < 0.357 \quad , \quad \frac{10m}{10k \times 0.357} < C$$

即 $C > \frac{1\mu}{0.357} \cong 2.8\mu F$

若抖动时间是20ms，则C>5.6μF，因此，C的值可定于2.8μF到5.6μF之间，笔者的习惯是R=10kΩ时，C采用3.3μF；若R为100kΩ，则C采用0.33μF。

放开按钮开关，开关弹开时，即将电容两端开路，使电容开始充电，当然电容两端电压不会立即升为高电平；而开关再弹回（短路）时，又将好不容易充电的电容两端短路。因此，电容两端电压在抖动期间，保持为低电平，不随抖动变化。直到抖动过后，电容两端的电压才稳定上升，丝毫不受抖动影响。这种方式简单又有效，所增加成本与电路复杂度都不高，称得上是实用的硬件防抖动电路。

● 软件防抖动

不管怎样，利用硬件来抑制抖动的噪声，一定会增加电路的复杂性与成本！我们只要在软件上下点功夫，避开产生抖动的那10~20ms，即可达到防抖动的效果。怎么做呢？只要在读入第一个状态的输入信号时，即执行10~20ms的延迟子程序（通常是20ms）即可。当按下按钮开关瞬间，程序将执行debouncer函数，而这个函数就是一个延迟子程序，如下所示。

```
void debouncer（void）          // 防抖动函数开始
{   int i;                      // 声明变量
    for（i=0; i<2400; i++）；   // 连数2 400次
}                               // 防抖动函数结束
```

debouncer函数

如图4-15所示，以产生负脉冲的按钮开关为例，当按下按钮，8051检测到第一个低电平信号时，随即调用debouncer函数以延迟20ms，这段时间程序不工作，以避开按钮开关上的不稳定状态。20ms后，程序才进行使用者按下按钮开关所应有的操作。同样地，当放开按钮，8051检测到第一个高电平信号时，随即调用debouncer函数以延迟20ms，这段时间程序不工作，以避开按钮开关上的不稳定状态。20ms后，程序才进行使用者放开按钮开关所应有的操作，如图4-15所示。

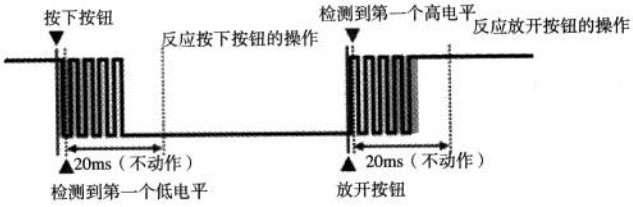


图4-15 按钮开关操作与防抖动函数的波形分析

通常只反应按钮开关的前沿，而不管后沿的变化。除非是要防止使用者按住按钮不放，如果一定要等到按钮放开，程序才要进行下一个操作的话，如图4-16所示，其操作分析如下。

1. 按下按钮，8051检测到第一个低电平信号时，随即调用debouncer函数以延迟20ms，这段时间程序不工作。

2. debouncer函数结束后，继续检测开关是否为高电平，若检测到第一个高电平，再调用debouncer函数以延迟20ms，这段时间程序不工作。

3. debouncer函数结束后，程序才进行该按钮所要进行的操作。

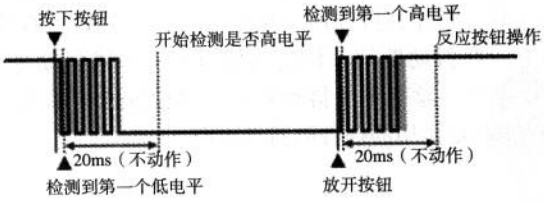


图4-16 放开按钮开关后才工作的波形分析

4.3 实例演练



在本单元里将针对按钮开关、指拨开关、数字型指拨开关，提供多个相关应用范例。

4.3.1 指拨开关控制实例演练

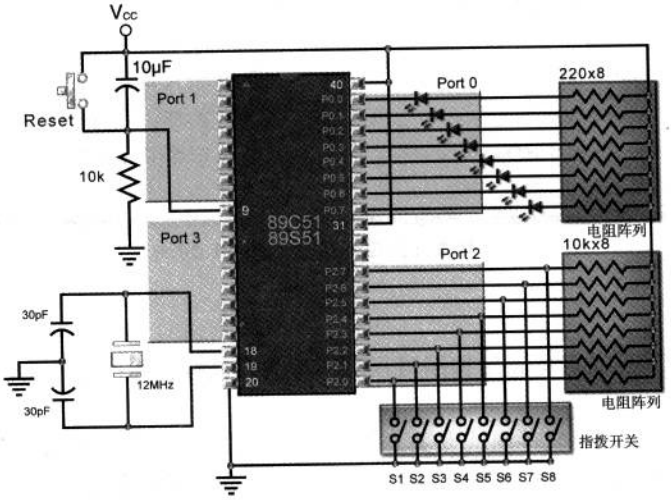


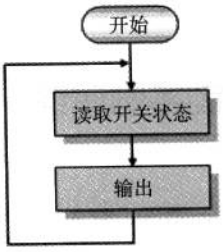
图4-17 指拨开关控制实验电路图

实验要点

就像家里墙壁上的开关一样，开关接通时，灯就会亮；切断开关时，灯就会灭。在此利用一个8P的指拨开关，当成8个墙壁开关，用来控制8个LED，如图4-17所示。指拨开关的状态由P2输入，而其状态将反应到P0所连接的LED上。若P2.0所连接的开关on，则P0.0所连接的LED将会亮、若P2.0所连接的开关off，则P0.0所连接的LED将不亮……，依次类推。

流程图与程序设计

根据功能需求与电路结构可知，当指拨开关on时，将可由其连接的输入口读取到低电平（即0）；而若要连接在P0的LED亮，则由P0输出低电平即可。因此，在程序里，只要将P2读取到的指拨开关直接输出到P0即可。




```
/* ch4_1.c - 基本开关控制实验 */
//==声明区==================================
#include      <reg51.h>                  // 定义8051寄存器的头文件，P2-17~19
#define SW    P2                         // 定义开关接至Port 2
```

```
#define LED    P0                          // 定义LED接至Port 0
//==主程序==================================
main()                                     // 主程序开始
{       SW=0xff;                           // 设计输入口
        while(1)                           // 无穷循环，程序一直运行
                LED=SW;                    // 读取开关(P2)状态，输出到LED(P0)
}                                          // 主程序结束
```

基本开关控制实验（ch4-1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图4-17连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，有没有“抖动”的困扰？

2. 若希望指拨开关中的S1、S3、S5三个开关都on，则前4个LED亮；S2或S4或S6开关on，则后4个LED亮；S7及S8开关on，则所有LED全亮，程序应如何编写？

3. 若将指拨开关换成一般家里墙壁上的开关，而LED换成继电器（RELAY），是否可作为家里的负载控制电路？

4.3.2 按钮ON-OFF控制实例演练

实验要点

如图4-18所示，若按一下PB1，则P0.0所连接的LED亮；若按一下PB2，则关闭P0.0所连接的LED（不亮）。

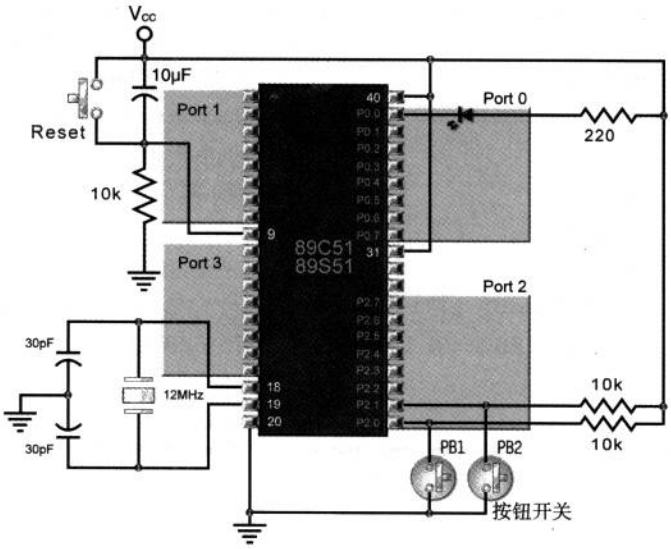
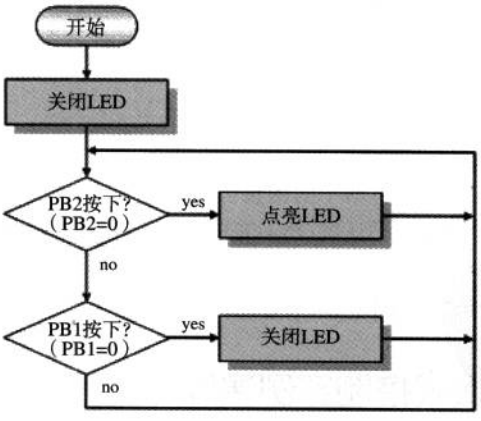


图4-18 按钮ON-OFF控制实验电路图

流程图与程序设计

根据功能需求与电路结构可知，当按下按钮开关时，将可由其连接的输入口读取到低电平（即0）；而若要连接在P0.0的LED亮，则由P0.0输出低电平即可。因此，在程序里，先声明P2.0为PB1、P2.1为PB2，接着关闭LED。然后判断PB1是否为0，若PB1=0，则点亮LED；若PB1不为0，再判断PB2是否为0，若PB2=0，则关闭LED，然后再从判断PB1开始，如此循环。




```
/* ch4_2.c -基本按钮 ON-OFF控制实验 */
//==声明区=========================================
#include<reg51.h>                    // 定义8051寄存器的头文件，P2-17~19
sbit PB1=P2^0;                       // 声明按钮1接至bit0(P2.0)
```

```
sbit  PB2=P2^1;                          // 声明按钮2接至bit1(P2.1)
sbit  LED=P0^0;                          // 声明LED为Port0的bit0
//==主程序=========================================
main()                                   // 主程序开始
{       LED=1;                           // 关闭LED
        PB1=PB2=1;   // 设计输入口
        while(1)                         // 无穷循环，程序一直运行
        {   if (PB1==0) LED=0;           // 若按下PB1，则点亮LED
            else if (PB2==0) LED=1;      // 若按下PB2，则关闭LED
        }                                // while循环结束
}                                        // 结束程序
```

基本按钮ON-OFF控制实验（ch4-2.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以产生*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图4-18连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，有没有“抖动”的困扰？

2. 若将按钮开关当成启动马达的ON-OFF开关，而LED换成继电器（RELAY），是否可作为马达控制？

3. 若同时按下PB1与PB2按钮会怎样？

4.3.3 按钮切换式控制实例演练

实验要点

如图4-19所示，若原本P0.0所连接的LED不亮，按一下PB1，则LED亮；再按一下PB1，则LED不亮……依次类推。

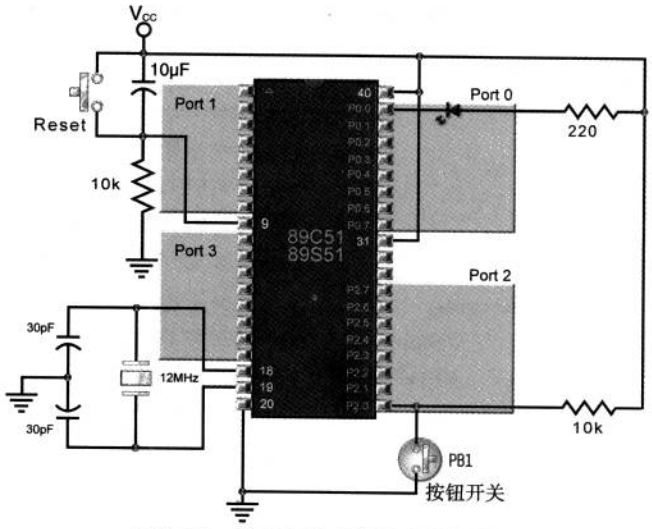
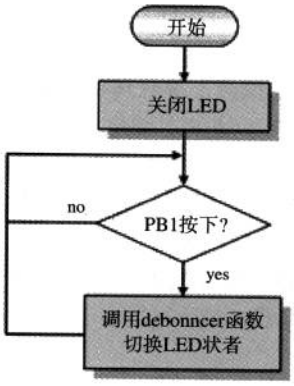


图4-19 按钮切换式控制实验电路图

流程图与程序设计

根据功能需求与电路结构可知，当按下PB1按钮开关时，即可改变LED的状态，若原本LED=1，按下PB1按钮开关即可使之变为0；若原本LED=0，按下PB1按钮开关即可使之变为1。这种情况最怕抖动，所以，一定要在检测到PB1被按下后，即进入防抖动函数，也就是延迟函数。同样地，在程序的开头，先声明相关的函数与变量，如LED、PB1、debouncer函数等，再关闭LED。然后判断PB1是否为0，若PB1=0，则调用debouncer函数，并切换LED状态。然后再从判断PB1开始，如此循环。




```
/* ch4_3.c - 按钮切换式控制实验 */
//==声明区=========================================
#include        <reg51.h>                    // 定义8051寄存器的头文件，P2-17~19
sbit  PB1=P2^0;                              // 声明PB1
```

```
sbit LED=P0^0;                          // 声明LED
void debouncer(void);  // 声明防抖动函数
//==主程序=====================================
main()                                  // 主程序开始
{       LED=1;                          // 关闭LED
        P2=0xff;                        // 设计P2为输入口
        while(1)                        // 无穷循环，程序一直运行
        {       if(PB1==0)              // 若按下PB1
                {       debouncer();    // 调用防抖动函数*/
                        LED=!LED;       // 切换LED为反相
                }                       // if语句结束
        }                               // while循环结束
}                                       // 主程序结束
//==子程序=====================================
/* 防抖动函数，延迟约20ms */
void debouncer(void)                    // 防抖动函数开始
{       int i;                          // 声明整数变量i
        for(i=0; i<2400; i++);          // 计数2 400次，延迟约20ms
}                                       // 防抖动函数结束
```

按钮切换式控制实验（ch4-3.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图4-19连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，改变debouncer函数的时间长短，看看会有什么影响？

2. 若按住PB1不放会怎样？如何改善？

4.3.4 按钮开关应用实例演练

实验要点

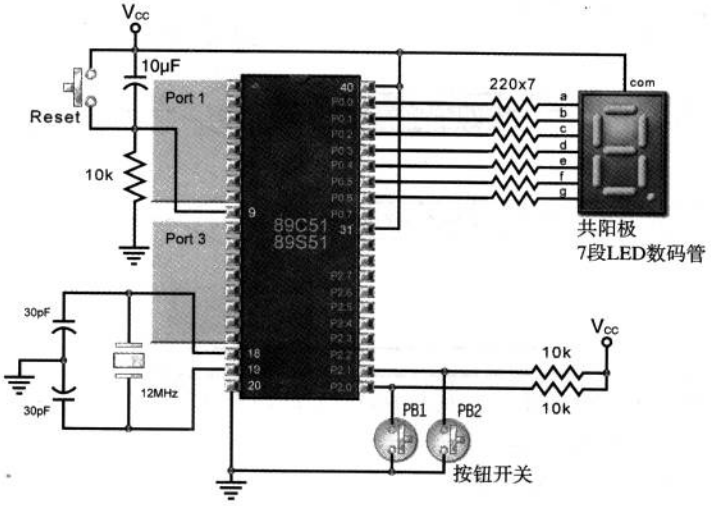


图4-20 按钮开关应用电路图

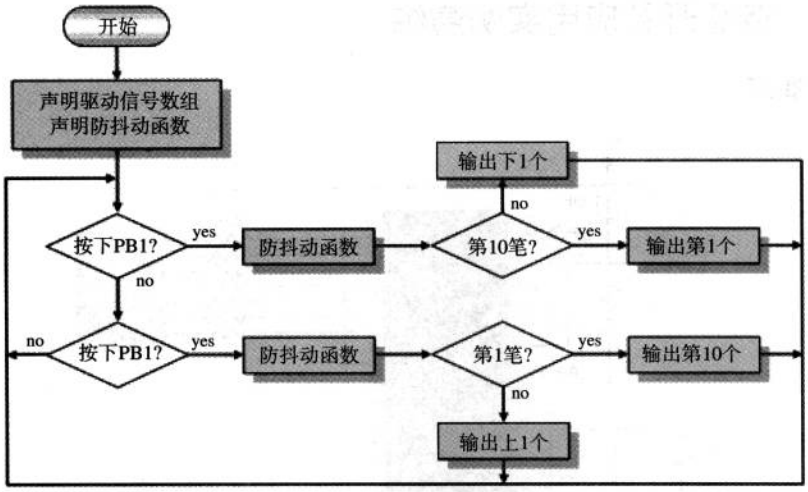
如图4-20所示，P0经限流电阻连接共阳极7段LED数码管。而P2.0连接PB1、P2.1连接PB2，其中PB1具有增数的功能，PB2具有减数的功能。若程序刚开始时，7段LED数码管显示0，按一下PB1，则7段LED数码管显示1、再按一下PB1，则7段LED数码管显示2……；若7段LED数码管显示9，按一下PB1，则7段LED数码管显示0。同样地，若7段LED数码管显示0，按一下PB2，则7段LED数码管显示9、再按一下PB2，则7段LED数码管显示8……依次类推。

流程图与程序设计

根据功能需求与电路结构可知，首先将共阳极7段LED数码管的驱动信号存为字符数组，即：

```
char TAB[10]={0xc0, 0xf9, 0xa4, 0xb0, 0x99,
              0x92, 0x83, 0xf8, 0x80, 0x98 };
```

主程序刚开始时，共阳极7段LED数码管显示0，也就是从数组中输出第一个驱动信号。然后判断PB1是否为0，若PB1=0，则输出下一个驱动信号，若超过10个数据，则从第1个数据开始；接着，判断PB2是否为0，若PB2=0，则输出上一个驱动信号，若原本是第1个数据，则输出第10个数据。




```
/* ch4_4.c -按钮开关应用(两按钮控制7段LED数码管加减数) - P4-21 */
//==声明区=============================================
#include        <reg51.h>                  // 定义8051寄存器的头文件，P2-17~19
#define SEG     P0                         // 定义7段LED数码管接至Port 0
/* 声明7段LED数码管驱动信号数组(共阳) */
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,    //数字0~4
               0x92, 0x83, 0xf8, 0x80, 0x98 }; //数字5~9
void debouncer(void);                     // 声明防抖动函数
sbit  PB1=P2^0;                           // 声明按钮1接至bit0(P2.0)
sbit  PB2=P2^1;                           // 声明按钮2接至bit1(P2.1)
//==主程序=============================================
main()                  // 主程序开始
{       unsigned char i=0;                // 声明变量i初值=0
        PB1=PB2=1;                        // 设计输入口
        while(1)                          // 无穷循环，程序一直运行
        {       SEG=TAB[i];               // 输出数字至7段LED数码管
                if (PB1==0)               // 判断PB1是否按下
                {   debouncer();          // 调用防抖动函数
                    i= (i<9)? i+1:0;      // 若i<9则i=i+1，若i>=9清除为0
                }                         // if语句结束
                if (PB2==0)               // 判断PB2是否按下
                {   debouncer();          // 调用防抖动函数
                    i=(i>0)? i-1:9;       // 若i>0则i=i-1，i<=0重设为9
```

```
                }                                    // if语句结束
            }                                        // while循环结束
}                                                    // 主程序结束
//==子程序=====================================
/* 防抖动函数，延迟约20ms */
void debouncer(void)                                 // 防抖动函数开始
{       int i;                                       // 声明整数变量i
        for(i=0; i<2400; i++);                       // 计数2 400次，延迟约20ms
}                                                    // 防抖动函数结束
```

按钮开关应用实验（ch4-4.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图4-20连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，若按钮按住不放会怎样？如何改善？

2. 在本实验里，若PB1与PB2两个按钮同时按会怎样？

4.3.5 BCD数字型指拨开关实例演练

实验要点

如图4-21所示，P0经限流电阻连接共阳极7段LED数码管，而P2的低四位连接到BCD数字型指拨开关，让BCD数字型指拨开关上的数字，显示在7段LED数码管里。

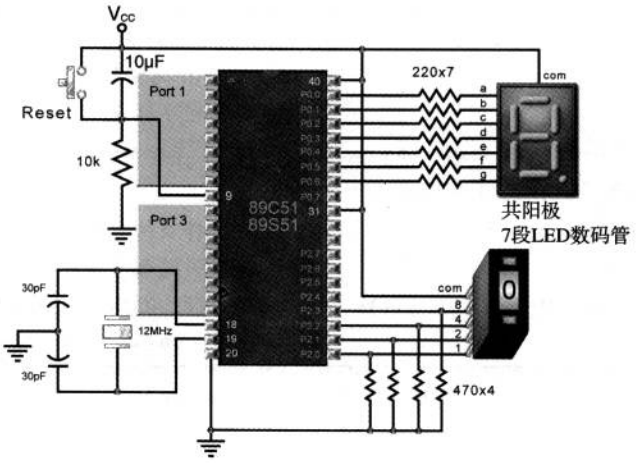
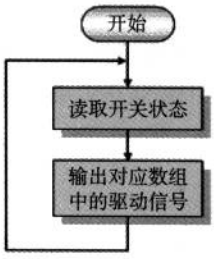


图4-21 BCD数字型指拨开关实验电路图

流程图与程序设计

根据功能需求与电路结构可知，首先将共阳极7段LED数码管的驱动信号存为字符数组，然后读入BCD数字型指拨开关的数字数据。再根据其将对应的数组数据，输出P0所连接的7段LED数码管即可。因此，本实验的程序只是读取P2，再将它送到P0而已，与4.3.1节的实验类似。




```
/* ch4_5.c - BCD数字型指拨开关实验 - P4-24 */
//==声明区=========================================
#include        <reg51.h>       // 定义8051寄存器的头文件，P2-17~19
#define SEG     P0              // 定义7段LED数码管接至Port 0
#define SW      P2              // 定义开关接至Port 2
/* 声明7段LED数码管驱动信号数组(共阳) */
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,            // 数字0~4
               0x92, 0x83, 0xf8, 0x80, 0x98 };          // 数字5~9
#define SW_H()SW&0x0f           // 读取开关值(P2清除高4bit)
//==主程序=========================================
main(                           // 主程序开始
{       SW=0xff;                // 设计输入口
```

```
    while(1)                    //无穷循环，程序一直运行
        SEG=TAB[SW_H()]; //读取开关值，输出至7段LED数码管(P0)
}                               //主程序结束
```

BCD数字型指拨开关实验（ch4-5.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图4-21连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，有没有“抖动”的困扰？

2. 若把本单元的BCD数字型指拨开关，改为16进制数字型指拨开关，程序应如何修改？

4.3.6 多重按钮开关实例演练

实验要点

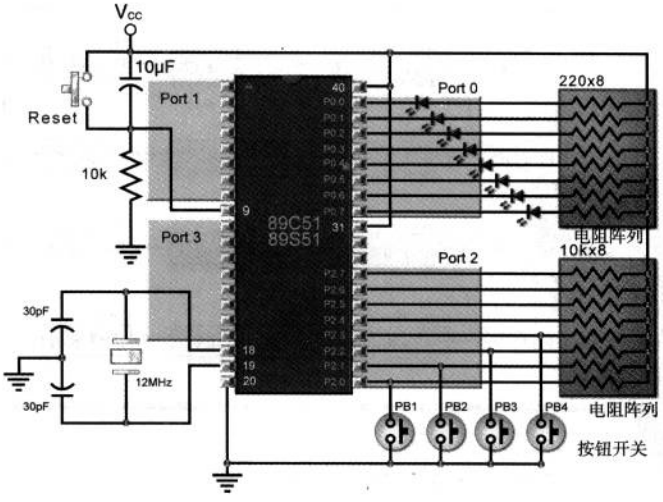


图4-22 多重按钮开关实验电路图

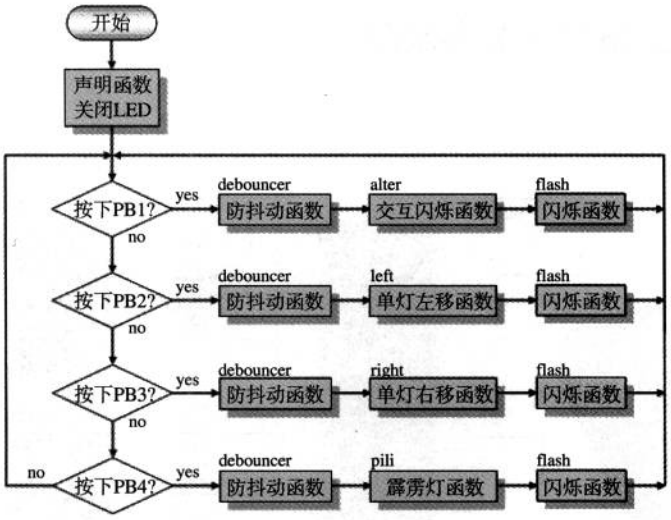
如图4-22所示，P0经限流电阻连接8个LED，而P2的低四位各连接一个按钮开关。当然，每个输入/输出端口上都通过10kΩ上拉电阻让它随时保持High。而本实验的功能说明如下。

- 按一下PB1钮，前4个LED、后4个LED 交互显示3次（即前4个LED亮、后4个LED不亮，0.5s后，切换为前4个LED不亮、后4个LED亮；如此重复三次），然后8个LED闪烁三次（每闪烁一次为全亮0.5s、全暗0.5s）。
- 按一下PB2钮，单灯左移3圈，然后8个LED闪烁3次。
- 按一下PB3钮，单灯右移3圈，然后8个LED闪烁3次。
- 按一下PB4钮，霹雳灯3圈，然后8个LED闪烁3次。

在本实验里有两个目的，第一个目的是凸显模块化的重要性，第二个目的是说明按钮的优先等级。因此，本实验将针对相同的电路、相同的功能，进行两个程序的实验。

流程图与程序设计之一

根据功能需求与电路结构可知，首先将所要执行的功能区分不同的函数，如防抖动函数、交互闪烁函数、单灯左移函数、单灯右移函数、霹雳灯函数、闪烁函数，还有延迟函数。再利用if-else if语句来判断PB1、PB2、PB3、PB4按钮是否被按下，再根据按钮状态，调用不同的函数，以执行它们的功能。至于这些函数，大都在前面的单元中已经操作过，在此只将它包装成函数的头文件myio.h，以供后续使用时能够直接调用，使整个程序变得很清楚！




```
/* myio.h - 自己写的链接库 */
//==声明区==============================
#define LED    P0                    // 定义LED接至Port 0
```

```
void debouncer(void);                    // 声明防抖动函数
void delay10ms(int);                     // 声明10ms延迟函数
void alter(int);                         // 声明交互闪烁函数
void left(int);                          // 声明单灯左移函数
void right(int);                         // 声明单灯右移函数
void pili(int);                          // 声明霹雳灯函数
void flash(int);                         // 声明闪烁函数
//==自己写的子程序=======================
/* 防抖动函数函数，延迟约20ms */
void debouncer(void)                     // 防抖动函数开始
{       int i;                           // 声明整数变量i
        for(i=0; i<2400; i++);           // 计数2400次，延迟约20ms
}                                        // 防抖动函数结束
/* 延迟函数开始，延迟约x×10ms */
void delay10ms(int x)                    // 延迟函数开始
{       int i, j;                        // 声明整数变量i, j
        for (i=0; i<x; i++)              // 计数x次，延迟约x×10ms
            for (j=1; j<=1200; j++);     // 计数1 200次，延迟约10ms
}                                        // 延迟函数结束
/* 高低位交互闪烁函数，执行x次 */
void alter(int x)                        // 高低位交互闪烁函数开始
{       int i;                           // 声明变量i
        LED=0x0f;                        // 初始状态(高位亮，低位灭)
        for(i=0; i<2*x-1; i++)           // i变量for循环执行2x-1次
        {       delay10ms(50);           // 延迟50*10m=0.5s
                LED=~LED;                // LED反相输出
        }                                // i变量for循环结束
        delay10ms(50);                   // 延迟50×10m=0.5s
}                                        // 高低位交互闪烁函数结束
/* 全灯闪烁函数，执行x次 */
void flash(int x)                        // 全灯闪烁函数开始
{       int i;                           // 声明变量i
        LED=0x00;                        // 初始状态(全亮)
        for(i=0; i<2*x-1; i++)           // i变量for循环执行2x-1次
        {       delay10ms(50);           // 延迟50×10m=0.5s
                LED=~LED;                // P0反相输出
        }                                // i变量for循环结束
        delay10ms(50);                   // 延迟50×10m=0.5s
```

```
}                                          // 全灯闪烁函数结束
/* 单灯左移函数，执行x圈 */
void left(int x)                           // 单灯左移函数开始
{       int i，j;                          // 声明变量i，j
        for(i=0；i<x；i++)                 // i循环，执行x圈
        {       LED=0xfe;                  // 初始状态=1111 1110
                for(j=0；j<7；j++)         // j循环，左移7次
                { delay10msv25);           // 延迟25×10m=0.25s
                  LED=(LED<<1)|0x01;       // 左移1位后，LSB设为1
                }                          // j循环结束
                delay10ms(25);             // 延迟25×10m=0.25s
        }                                  // i循环结束
}                                          // 单灯左移函数结束
/* 单灯右移函数，执行x圈 */
void right(int x)                          // 单灯右移函数开始
{       int i，j;                          // 声明变量i，j
        for(i=0；i<x；i++v                 // i循环，执行x圈
        {       LED=0X7f;                  // 初始状态=0111 1111
                for(j=0；j<7；j++)         // j循环，右移7次
                { delay10ms(25);           // 延迟25×10m=0.25s
                  LED=(LED>>1)|0x80;       // 左移1位后，MSB设为1
                }                          // j循环结束
                delay10ms(25v;             // 延迟25×10m=0.25s
        }                                  // i循环结束
}                                          // 单灯左移函数结束
/* 霹雳灯函数，执行x圈 */
void pili(int x)                           // 霹雳灯函数开始
{       int i;                             // 声明变量i
        for(i=0；i<x；i++)                 // i循环，执行x圈
        {       left(1);                   // 单灯左移一圈
                right(1);                  // 单灯左移一圈
        }                                  // i循环结束
}                                          // 霹雳灯函数结束
```

自己编写的链接库（myio.h）

```
/* ch4_6_1.c - 多重按钮开关实验之1 - P4-26 */
//==声明区==================================
```

```
#include      <reg51.h>                  // 定义8051寄存器的头文件，P2-17~19
#include      "myio.h"                   // 自己写的I/O链接库
sbit PB1=P2^0;                           // 声明PB1
sbit PB2=P2^1;                           // 声明PB2
sbit PB3=P2^2;                           // 声明PB3
sbit PB4=P2^3;                           // 声明PB4
//==主程序==================================
main()                                   // 主程序开始
{       LED=0xff;                        // 初始状态(LED全灭)
        P2=0xff;                         // 设计P2输入口
        while(1)                         // 无穷循环，程序一直运行
        {       if (PB1==0v)             // 如果按下PB1
                {   debouncer();         // 防抖动
                    alter(3);            // 高低位交互闪烁3次
                    flash(3); }          // 全灯闪烁3次
                else if (PB2==0)         // 如果按下PB2
                {   debouncer();         // 防抖动
                    left(3);             // 单灯左移3圈
                    flash(3); }          // 全灯闪烁3次
                else if vPB3==0)         // 如果按下PB3
                {   debouncer();         // 防抖动
                    right(3);            // 单灯右移3圈
                    flash(3); }          // 全灯闪烁3次
                else if (PB4==0)         // 如果按下PB4
                {   debouncer();         // 防抖动
                    pili(3);             // 霹雳灯3圈
                    flash(3); }          // 全灯闪烁3次
        }                                // while循环结束
}                                        // 主程序结束
```

多重按钮开关实验之一（ch4-6-1.c）

操作之一

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图4-22连接线路，使用在线仿真器，加载新的程

序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，若同时按下多个按钮会如何？

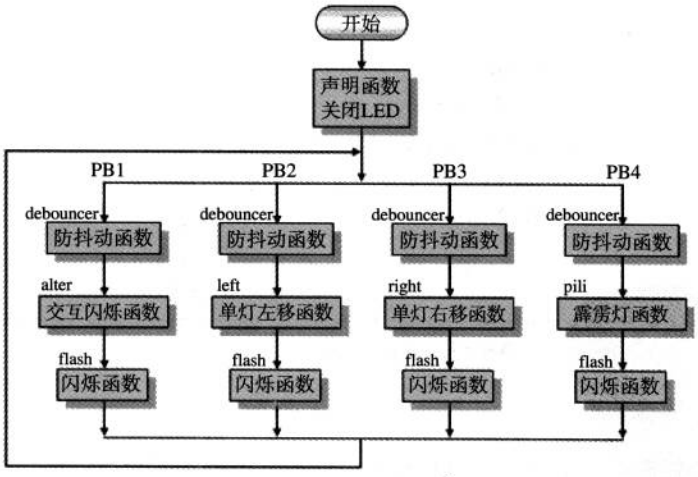
2. 在本实验里，若按住按钮不放会如何？

3. 在本实验里，其中debouncer函数是个延迟20ms的函数，而delay10ms函数是个延迟10ms的函数，可否使用delay10ms函数取代debouncer函数？如何修改？

4. 在本实验里，其中alter函数是个高四位与低四位交互闪烁的函数，而flash函数是个8灯闪烁的函数，其不同在于其初始值。请修改函数，再增加一个传入自变量，以传入初始值，使它变成一个多用途的函数？

流程图与程序设计之二

在前面的实验里，各按钮之间有优先等级的问题，PB1具有最高优先等级，其次是PB2……依次类推。若改成无优先权的问题时，可利用switch语句，取代if-else if语句。




```
/* ch4_6_2.c - 多重按钮开关实验之2 - P4-30 */
//==声明区=========================================
#include      <reg51.h>          // 定义8051寄存器的头文件，P2-17~19
#include      "myio.h"           // 自己写的I/O链接库 P4-26
```

```
#define PB      P2                          // 定义按钮开合接至Port 2
//==主程序=========================================
main()                                      // 主程序开始
{       LED=0xff;                           // 初始状态(LED全灭)
        PB=0xff;                            // 设计输入口
        while(1)                            // 无穷循环，程序一直运行
                switch(~PB)                 // switch语句开始
                {   case 0x01:              // 如果按下PB1
                    {   debouncer();        // 防抖动
                        alter(3);           // 交互闪烁3次
                        flash(3);           // 全灯闪烁3次
                        break; }            // 退出switch语句
                    case 0x02:              // 如果按下PB2
                    {   debouncer();        // 防抖动
                        left(3);            // 单灯左移3圈
                        flash(3);           // 全灯闪烁3次
                        break; }            // 退出switch语句
                    case 0x04:              // 如果按下PB3
                    {   debouncer();        // 防抖动
                        right(3);           // 单灯右移3圈
                        flash(3);           // 全灯闪烁3次
                        break; }            // 退出switch语句
                    case 0x08:              // 如果按下PB4
                    {   debouncer();        // 防抖动
                        pili(3 ) ;          // 霹雳灯3圈
                        flash(3);           // 全灯闪烁3次
                        break; }            // 退出switch语句
                }                           // 结束switch语句*/
}                                           // 主程序结束
```

多重按钮开关实验之二（ch4-6-2.c）

操作之二

1. 按上述流程图与程序设计，重新设计程序，并重复执行调试与仿真。

2. 若软件调试/仿真功能正常，再使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

3. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

4. 编写实验报告，并语句这两个程序的差异。

思考一下

1. 在本实验里，若同时按下多个按钮会如何？
2. 在本实验里，若按住按钮不放会如何？

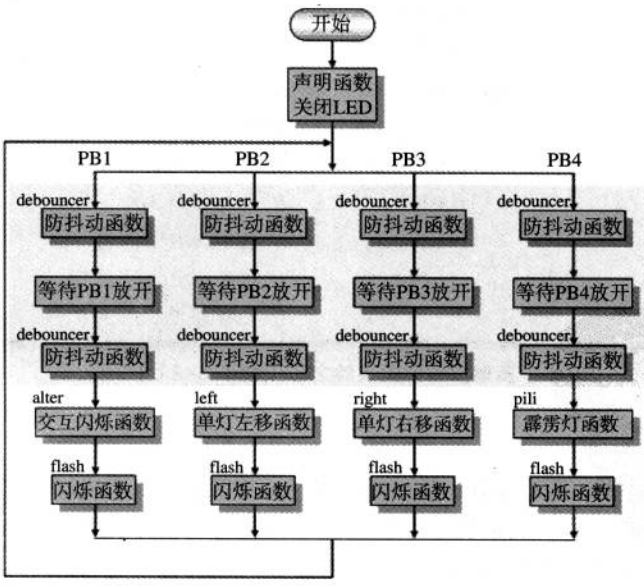
4.3.7 按钮开关放开后动作实例演练

实验要点

本实验接续4.3.6节的实验，所使用相同的电路图、相同的功能要求，但在此将解决按钮按住不放的窘境。

流程图与程序设计

根据功能需求与电路结构可知，在此将从检测到按钮的第一个低电平（按下按钮瞬间）开始，先延迟20ms，再检测该按钮是否为高电平；同样地，当该按钮变为高电平（即放开按钮瞬间），再延迟20ms，让按钮完全稳态后，才执行该按钮所付予的任务。整个按钮的操作分析详见图4-16。延续4.3.6节中以switch语句方式所编写的程序，在此将原本单纯的检测到按钮的第一个低电平后，执行防抖动函数，更改（插入）为执行防抖动函数后，再检测按钮是否为高电平，若检测到按钮的第一个高电平，再执行一次防抖动函数，如下流程图中蓝底的部分为更改（插入）的部分。




```
/* ch4_7.c - 按钮放开后才工作的实验 - P4-34 */
//==声明区=========================================
```

```
#include<reg51.h>                    // 定义8051寄存器的头文件，P2-17~19
#include        "myio.h"             // 自己写的I/O链接库
sbit PB1=P2^0;                       // 声明PB1
sbit PB2=P2^1;                       // 声明PB2
sbit PB3=P2^2;                       // 声明PB3
sbit PB4=P2^3;                       // 声明PB4
//==主程序=======================================
main()                               // 主程序开始
{       LED=0xff;                    // 初始状态(LED全灭)
        P2=0xff;                     // 设计P2为输入口
        while(1)                     // 无穷循环，程序一直运行
            switch(~P2)              // switch语句开始
            {   case 0x01:           // 如果按下PB1
                {   debouncer();     // 防抖动
                    while(PB1==0);   // 等待PB1放开
                    debouncer();     // 防抖动
                    alter(3);        // 交互闪烁3次
                    flash(3);        // 全灯闪烁3次
                    break; }         // 退出switch语句
                case 0x02:           // 如果按下PB2
                {   debouncer();     // 防抖动
                    while(PB2==0);   // 等待PB2放开
                    debouncer();     // 防抖动
                    left(3);         // 单灯左移3圈
                    flash(3);        // 全灯闪烁3次
                    break; }         // 退出switch语句
                case 0x04:           // 如果按下PB3
                {   debouncer();     // 防抖动
                    while(PB3==0);   // 等待PB3放开
                    debouncer();     // 防抖动
                    right(3);        // 单灯右移3圈
                    flash(3);        // 全灯闪烁3次
                    break; }         // 退出switch语句
                case 0x08:           // 如果按下PB4
                {   debouncer();     // 防抖动
                    while(PB4==0);   // 等待PB4放开
                    debouncer();     // 防抖动
                    pili(3);         // 霹雳灯3圈
```

```
                    flash(3);          // 全灯闪烁3次
                    break; }           // 退出switch语句
                }                      // 结束switch语句
}                                      // 主程序结束
```

放开按钮后工作的实验（ch4-7.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图4-22连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的操作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

在本实验里，若将按钮按住不放会如何？

4.4 实时练习



在本章里探讨了8051的输入端口相关的主题以及输入设备的介绍与应用。在程序设计上，应用了不同的语句与指令以达到类似的功能，并处理输入时可能的状态，可说是数字与微处理系统中，相当重要的一部分。在此请试着回答下列问题，以确认对于此部分的掌握程度。

选择题

（　）1. 在8051的程序里，若要将某个输入/输出端口设计成输入功能，应如何处理？
（A）先输出高电平到该输入/输出口　（B）先输出低电平到该输入/输出口
（C）先读取该输入/输出口的状态　（D）先储存该输入/输出口的状态

（　）2. 下列哪种开关具有自动回复功能？
（A）指拨开关　（B）闸刀开关　（C）摇头开关　（D）按钮开关

（　）3. 下列哪种开关具有多输出状态？

（A）摇头开关　　（B）TACK switch
（C）BCD数字型指拨开关　（D）以上皆非

（　）4. 若要产生边缘触发信号，通常会使用哪种开关？
（A）指拨开关 （B）闸刀开关 （C）按钮开关 （D）数字型指拨开关

（　）5. 通常电路板上的厂商设定/调整，可使用哪种开关？
（A）指拨开关 （B）闸刀开关 （C）按钮开关 （D）数字型指拨开关

（　）6. 根据实验统计，当操作开关时其不稳定状态大约会持续多久？
（A）1~5ms （B）10~20ms （C）100~150ms （D）150~250μs

（　）7. 在电路板上的跳线（Jumper）常被哪种开关替代？
（A）指拨开关 （B）闸刀开关 （C）按钮开关 （D）数字型指拨开关

（　）8. 在Keil C里，判读开关状态使用if-else if语句与使用switch语句有何差异？
（A）if-else if语句较快　　（B）if-else if语句有优先级
（C）switch语句可判读较多开关状态　（D）switch语句有优先级

（　）9. 下列何者不是数字型指拨开关？
（A）16进制数字型指拨开关　（B）BCD数字型指拨开关
（C）12进制数字型指拨开关　（D）以上皆是

（　）10. 对于低电平工作（低电平触发）的开关而言，下列何者不是在输入口上连接一个上拉电阻到V_{cc}的目的？
（A）提供足够的驱动电流　（B）防止不确定状态
（C）保持输入高电平　　（D）防止感染噪声

问答题

1. 在8051里，若输入/输出端口执行输入功能之前，为何要先送“1”到该输入/输出口？

2. 试述如何使用BCD数字型指拨开关？其输出信号为何？

3. 常用的开关可分为按钮开关及单刀开关两种，若要取得脉冲信号，应使用哪一种开关？若要取得电平信号，应使用哪一种开关？而指拨开关属于哪一种开关？

4. 若在8051里使用了开关作为输入设备，试说明如何在开关设备RC电路，即可防抖动？

5. 什么是“抖动”？请绘制一个低电平工作的开关波形分析图？

6. 在程序里，如何以简单的方式来防止输入开关的抖动现象？

7. 为了避免使用者按住按钮不放所造成错误或不确定状态，请画出解决这种状况的流程或操作？

（A）指针开关　（B）旋钮开关　（C）拨动开关　（D）按钮开关

（　）2. 晶体管时基电路 NE555 的输出端可以直接驱动（　　）。

（A）发光二极管　（B）扬声器　（C）继电器　（D）以上都可以

（　）3. 晶体管时基电路 NE555 的输出端输出的是（　　）。

（A）模拟信号　（B）数字信号　（C）脉冲信号　（D）以上都不是

（　）4. 晶体管时基电路 NE555 的输出端输出的信号是由（　　）决定的。

（A）输入信号　（B）电源电压　（C）电阻电容　（D）以上都不是

（　）5. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

（A）模拟信号　（B）数字信号　（C）脉冲信号　（D）以上都不是

（　）6. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

（A）模拟信号　（B）数字信号　（C）脉冲信号　（D）以上都不是

（　）7. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

（A）模拟信号　（B）数字信号　（C）脉冲信号　（D）以上都不是

（　）8. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

（A）模拟信号　（B）数字信号　（C）脉冲信号　（D）以上都不是

（　）9. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

（A）模拟信号　（B）数字信号　（C）脉冲信号　（D）以上都不是

（　）10. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

（A）模拟信号　（B）数字信号　（C）脉冲信号　（D）以上都不是

二、填空题（每空 2 分，共 20 分）

1. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

2. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

3. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

4. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

5. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

6. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

7. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

8. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

9. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

10. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

三、简答题（每题 10 分，共 30 分）

1. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

2. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

3. 晶体管时基电路 NE555 的输出端输出的信号是（　　）。

第5章 输入/输出口的高级应用

本章内容丰富，主要包括以下两部分。

硬件部分：

介绍4×4键盘、7段LED数码管模块、7447、74138等。

程序与实践部分：

键盘扫描程序的应用。

7段LED数码管扫描程序的应用。

5.1 键盘扫描



在第4章里，我们使用了4个按钮开关，结果占用了4个位的输入/输出端口。如果以同样的方式连接16个按钮的话，就得占用16位的输入/输出端口，这绝对不是好方法！对于数字/微型计算机电路而言，若需要多个按钮，通常会将这些按钮组成数组，例如16个按钮，则排列成4×4数组，称之为键盘（Keypad），如图5-1所示。

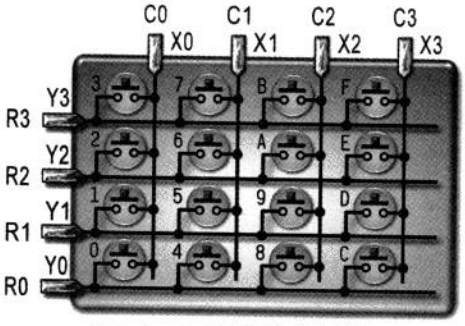


图5-1 4x4键盘的内部结构

所谓“4×4”是指4行（Column）与4列（Row）所构成的按键数组。为了说明方便起见，由下而上各列编制为R0、R1、R2及R3，由左而右各行编制为C0、C1、C2及C3，而每个按键按照顺序编制为0~9、A~F。由于列属于Y轴，也可将R0、R1、R2及R3改为Y0、Y1、Y2及Y3；而行属于X轴，也可将C0、C1、C2及C3，改为X0、X1、X2及X3。当然，也可以随使用者的需要自行编制。在电路设计上，按钮可以使用Tack Switch。如下图所示为常用的Tack Switch，其外表是一个具有4个引脚的正方形，而其内部是将两对引脚内部连接，两对引脚之间则为a接点，如图5-2所示。

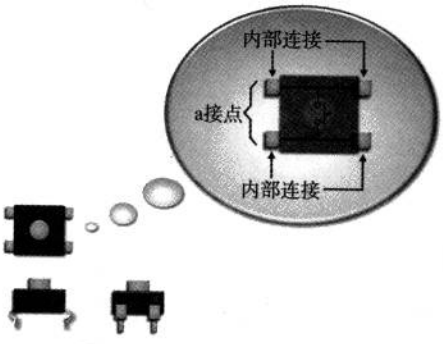


图5-2 Tack Switch的结构

使用这种具有内部连接的Tack Switch，则可在单面电路板（或面包板）上轻松制作4×4键盘，如图5-3所示。

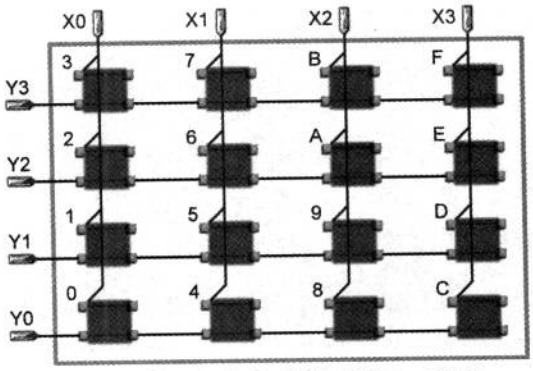


图5-3 由Tack Switch所构成的4×4键盘

当然，这样还不够，在每列上都必须连接一个上拉电阻（10kΩ即可），我们可使用一排4R5P的电阻，比较省事，如图5-4所示。

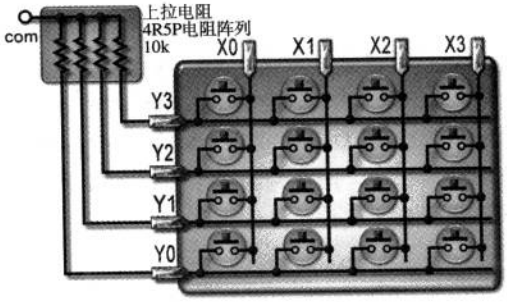
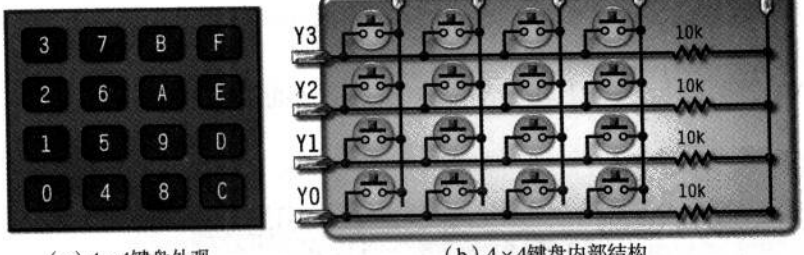


图5-4 连接上拉电阻

某些现成的键盘已将上拉电阻放置在其中。在本章中将把上拉电阻看作键盘的一部分，而不进行刻意说明，如图5-5所示。



（a）4×4键盘外观　　（b）4×4键盘内部结构

图5-5 4x4键盘

另外，在市场上也可以买到4×4键盘成品（如图5-6所示），相当便宜，但是其按键的编号不见得适用，而其中不含上拉电阻，必须再连接一排电阻。

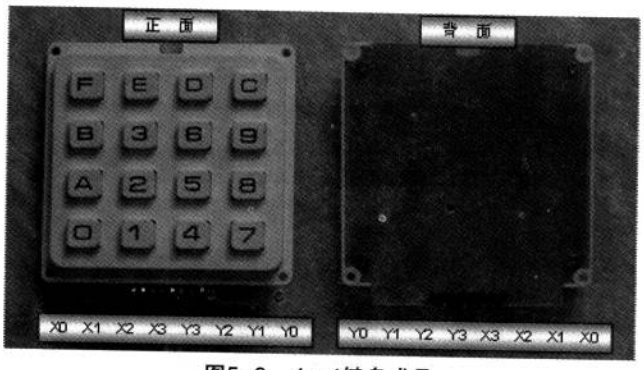


图5-6　4×4键盘成品

5.1.1　键盘扫描原理

如图5-5（a）所示为4×4键盘，而（b）图为其内部结构，其中包含4行、4列，构成一个4×4的数组，在此将每行连接端点定名为X0、X1、X2及X3，而每列连接端点定名为Y0、Y1、Y2及Y3。另外，每列各连接一个电阻到共同接点（com）上。根据扫描方式的不同，com可能连接到V_{cc}或GND，当我们要进行键盘扫描时，则将扫描信号送至X0~X3，再由Y0~Y3读取键盘状态，即可判断哪个按键被按下。键盘扫描的方式有两种，即低电平扫描与高电平扫描，说明如下。

● 低电平扫描

低电平扫描是将共同点com连接V_{cc}，没有按任何按键被按下时，Y3、Y2、Y1、Y0端点能保持为高电平（即1）。送入X3、X2、X1、X0的扫描信号中，只有一个为低电平（即0），其余3个为高电平。整个工作可分为4个阶段，说明如下。

1. 在第一阶段，主要目的是判断key3、key2、key1及key0有没有被按下。首先将1110B信号送入X3、X2、X1、X0，也就是只有X0为低电平，其他各行均为高电平。接着读取Y3、Y2、Y1、Y0的状况。

• 若Y3、Y2、Y1、Y0为1110，表示key0被按下，如图5-7所示。

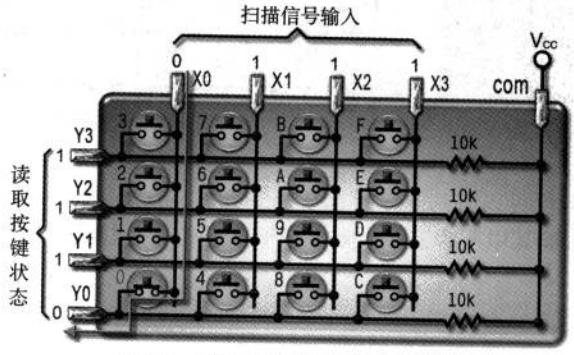


图5-7　低电平扫描－按下“0”键

• 若Y3、Y2、Y1、Y0为1101，表示key1被按下。

• 若Y3、Y2、Y1、Y0为1011，表示key2被按下。

• 若Y3、Y2、Y1、Y0为0111，表示key3被按下。

若Y3、Y2、Y1、Y0为1111，表示key3、key2、key1及key0都没被按下，进入第二个阶段。

2. 在第二阶段，主要目的是判断key7、key6、key5及key4有没有被按下。首先将1101B信号送入X3、X2、X1、X0，也就是只有X1为低电平，其他各行均为高电平。接着读取Y3、Y2、Y1、Y0的状态。

• 若Y3、Y2、Y1、Y0为1110，表示key4被按下。

• 若Y3、Y2、Y1、Y0为1101，表示key5被按下。

• 若Y3、Y2、Y1、Y0为1011，表示key6被按下。

• 若Y3、Y2、Y1、Y0为0111，表示key7被按下。

若Y3、Y2、Y1、Y0为1111，表示key7、key6、key5及key4都没被按下，进入第三个阶段。

3. 在第三阶段，主要目的是判断keyB、keyA、key9及key8有没有被按下。首先将1011B信号送入X3、X2、X1、X0，也就是只有X2为低电平，其他各行均为高电平。接着读取Y3、Y2、Y1、Y0的状态。

• 若Y3、Y2、Y1、Y0为1110，表示key8被按下。

• 若Y3、Y2、Y1、Y0为1101，表示key9被按下。

• 若Y3、Y2、Y1、Y0为1011，表示keyA被按下。

• 若Y3、Y2、Y1、Y0为0111，表示keyB被按下。

若Y3、Y2、Y1、Y0为1111，表示keyB、keyA、key9及key8都没被按下，进入第四个阶段。

4. 在第四阶段，主要目的是判断keyF、keyE、keyD及keyC有没有被按下。首先将0111B信号送入X3、X2、X1、X0，也就是只有X3为低电平，其他各行均为高电平。接着读取Y3、Y2、Y1、Y0的状态。

• 若Y3、Y2、Y1、Y0为1110，表示keyC被按下。

• 若Y3、Y2、Y1、Y0为1101，表示keyD被按下。

• 若Y3、Y2、Y1、Y0为1011，表示keyE被按下。

• 若Y3、Y2、Y1、Y0为0111，表示keyF被按下。

若Y3、Y2、Y1、Y0为1111，表示keyF、keyE、keyD及keyC都没被按下，进入第一个阶段，从头开始，继续扫描。

● 高电平扫描

高电平扫描是将共同点com接地（GND），没有按任何按键被按下时，Y3、Y2、Y1、Y0端点能保持为低电平（即0）。送入X3、X2、X1、X0的扫描信号之中，只有一个为高电平（即1），其余三个为低电平。整个工作可分为四个阶段，说明如下。

1. 在第一阶段，主要目的是判断key3、key2、key1及key0有没有被按下。首先将

0001B信号送入X3、X2、X1、X0，也就是只有X0为高电平，其他各行均为低电平。接着读取Y3、Y2、Y1、Y0的状态：

- 若Y3、Y2、Y1、Y0为0001，表示key0被按下，如图5-8所示。

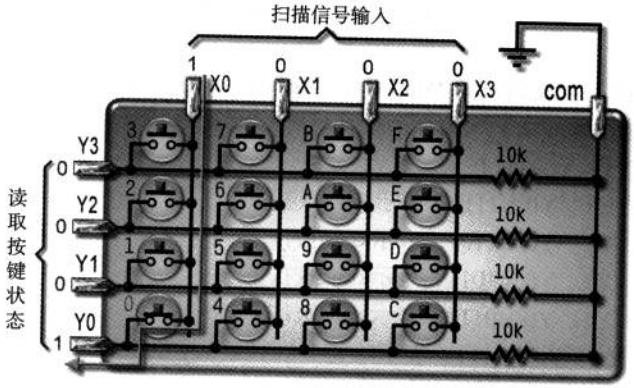


图5-8　高电平扫描－按下“0”键

- 若Y3、Y2、Y1、Y0为0010，表示key1被按下。
- 若Y3、Y2、Y1、Y0为0100，表示key2被按下。
- 若Y3、Y2、Y1、Y0为1000，表示key3被按下。

若Y3、Y2、Y1、Y0为0000，表示key3、key2、key1及key0都没被按下，进入第二个阶段。

2. 在第二阶段，主要目的是判断key7、key6、key5及key4有没有被按下。首先将0010B信号送入X3、X2、X1、X0，也就是只有X1为高电平，其他各行均为低电平。接着读取Y3、Y2、Y1、Y0的状态。

- 若Y3、Y2、Y1、Y0为0001，表示key4被按下。
- 若Y3、Y2、Y1、Y0为0010，表示key5被按下。
- 若Y3、Y2、Y1、Y0为0100，表示key6被按下。
- 若Y3、Y2、Y1、Y0为1000，表示key7被按下。

若Y3、Y2、Y1、Y0为0000，表示key7、key6、key5及key4都没被按下，进入第三个阶段。

3. 在第三阶段，主要目的是判断keyB、keyA、key9及key8有没有被按下。首先将0100B信号送入X3、X2、X1、X0，也就是只有X2为高电平，其他各行均为低电平。接着读取Y3、Y2、Y1、Y0的状态。

- 若Y3、Y2、Y1、Y0为0001，表示key8被按下。
- 若Y3、Y2、Y1、Y0为0010，表示key9被按下。
- 若Y3、Y2、Y1、Y0为0100，表示keyA被按下。
- 若Y3、Y2、Y1、Y0为1000，表示keyB被按下。

若Y3、Y2、Y1、Y0为0000，表示keyB、keyA、key9及key8都没被按下，进入第四个阶段。

4. 在第四阶段，主要目的是判断keyF、keyE、keyD及keyC有没有被按下。首先将

1000B信号送入X3、X2、X1、X0，也就是只有X3为高电平，其他各行均为低电平。接着读取Y3、Y2、Y1、Y0的状态。

- 若Y3、Y2、Y1、Y0为0001，表示keyC被按下。
- 若Y3、Y2、Y1、Y0为0010，表示keyD被按下。
- 若Y3、Y2、Y1、Y0为0100，表示keyE被按下。
- 若Y3、Y2、Y1、Y0为1000，表示keyF被按下。

若Y3、Y2、Y1、Y0为0000，表示keyF、keyE、keyD及keyC都没被按下，进入第一个阶段，从头开始继续扫描。

或许你会质疑，若在第一阶段扫描时，有人按key0、key1、key2、key3以外的按键，是不是就检测不到了？不必担心，由于人类手指的动作很慢，按下按键到放开按键的时间，至少也得0.1s（即100ms），而CPU的操作是以μs计算的，从第一阶段到第四阶段跑一圈，只需要几ms（ms）而已，所以，手都还没放开，程序就不知道扫描过多少次了！另外，通常以低电平扫描承居多，在本章中也将以低电平扫描为例，以讨论其操作与程序分析。

5.1.2 4×4键盘扫描程序解析

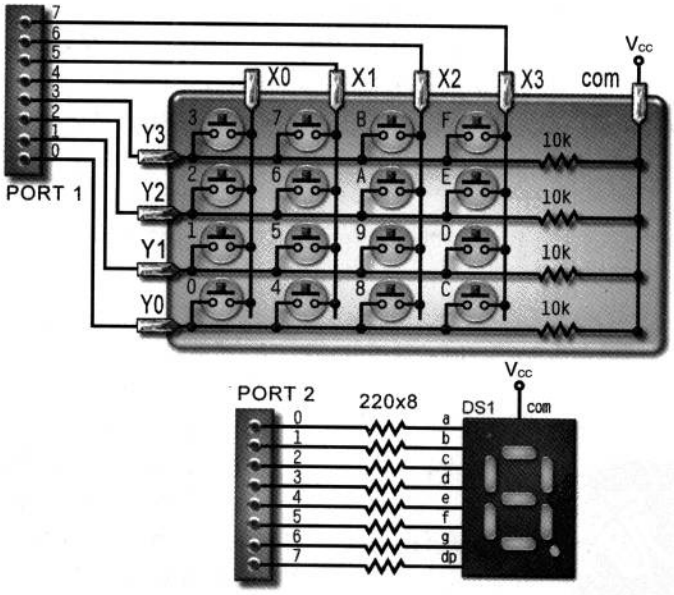


图5-9 4×4键盘扫描电路

如图5-9所示，当我们按下键盘里的按键后，按键上的键值将显示在DS17段LED数码管上。在介绍键盘扫描程序之前，必须先准备好16个7段LED数码管的驱动信号编码，除了第3章曾经使用过的0~9外，还要准备a~f，如表5-1所示。

表5-1

数字	（dp）gfedcba	16进制	显示
a	10100000	0xa0	
b	10000011	0x83	
c	10100111	0xa7	
d	10100001	0xa1	
e	10000100	0x84	
f	10001110	0x8e	

其中的 与6的编码一样，所以将原本6的编码，由0x83改为0x82，即 。

在此我们将利用两个for语句的巢状循环来判断按键，并写成一个scan函数，其中也调用了debouncer函数以防抖动，但debouncer函数的内容就不列出来。一开始没有任何按下时，显示小数点；按下按键时，显示键值；若出现同时按下多个键等错误状态，显示负号，说明如下。

```
/* 声明7段LED数码管驱动信号数组(共阳) */
char TAB[18]={ 0xc0,  0xf9,  0xa4,  0xb0,  0x99,     // 数字0-4
               0x92,  0x82,  0xf8,  0x80,  0x98,     // 数字5-9
               0xa0,  0x83,  0xa7,  0xa1,  0x84,     // 字母a-e(10-14)
               0x8e,  0xbf,  0x7f };  // 字母F(15), 负号(-), 小数点(.)
char disp = 0x7f ;                    // 声明7段LED数码管初值为小数点
:
:
#define  rowkey()  (~P1)&0x0f          //读入P1低4位(列按键值)宏
:
:
/* 扫描4×4键盘及4个7段LED数码管函数 */
void scanner(void)                    // 扫描函数开始
{    char col, row;                   // 声明变量(目前扫描第col行, 第row列)
     char scan, keyin, kcode;         // 声明变量(扫描信号, 列按键值, 扫描码)
     can=0xef;                        // 扫描信号初值=1110 1111
     for(col=0; col<4; col++)         // for循环, 扫描第col行
     {   P2=0xff;                     // 关闭7段LED数码管
         P1=scan;
         // P1高4位输出扫描信号, 低4位设为输入
         P2=disp;                     // 显示按键值
         keyin=rowkey();              // 读入P1低4位(第col行的列按键值)
         if(keyin != 0)               // 若有按键时
         {   for(row=0; row<4; row++) // 扫描第row列是否按下
```

```
            {   if(keyin ==(0x01<<row))          // 只有第row列按下
                { kcode=row+4*col;     // 计算出按键码为
                  disp=TAB[kcode];     // 扫描码写入7段LED数码管
                  break;               // 离开第row列扫描(for循环)
                 }                     // if语句(第row列按下)结束
                  if (row==4)          // 输入错误时
                  disp=TAB[16];        // 负号(-)写入7段LED数码管数组
        }                              // for循环，扫描row列结束
          while(rowkey() !=0);         // 等按钮放开
          debouncer();                 // 防抖动函数
       }                               // if语句(有按键时)结束
     scan=(scan<<1)|0x01;              // 产生左1位扫描信号
    }                                  // for循环，扫描col行结束
}                                      // 扫描函数结束
```

- “#define rowkey() (~P1)&0x0f”预处理的功是将程序之中的“rowkey()”，改为用“(~P1) &0x0f”宏替代，也就是将P1读入的按键状态反相，再与0x0f进行AND运算，其目的是屏蔽高四位（使之为0），而低四位（我们所要的按键状态）保持原状。“rowkey ()”比“(~P1) &0x0f”容易理解，程序也比较有水平！
- “keyin=rowkey () ; ”指令相当于“keyin= (~P1) &0x0f; ”，也就是把读入的按键状态，将其高四位变为0后，放入keyin变量，若keyin不等于0，表示有按键被按下。换言之，就是读取按键状态，处理后放入keyin变量。
- 在“if (keyin == (0x01<<row))”指令中，“(0x01<<row)”是把0x01左移row位，也就是判断第row列，就先把0x01左移row位。若输入的按键状态与0x01左移row位的状态一样，表示该列的按键被按下。
- “kcode=row+4*col; ”指令是将当时扫描的列（row）值，加上4倍的行（col）值，作为定位按键的方法。例如目前扫描第0行（col=0）和第2列（row=2），则可定位出编号为2的按键被按下（即2+4 × 0）；若目前扫描第2行（col=2）和第1列（row=1），则可定位出编号为9的按键被按下（即1+4 × 2）。
- “disp=TAB[kcode]; ”指令是将按键编号转换成所要显示的数值编码，并放入disp变量。
- “while (rowkey () !=0) ; ”指令是等按键放开后，才继续下面的操作。当然，放开按键也有抖动的问题，所以在其下有个“debouncer () ; ”防抖动指令。
- “scan= (scan<<1) |0x01; ”指令是将扫描码左移一位，而其最右边填入1。例如原本扫描码为“11111110”，左移一位变成“11111100”。再与0x01进行OR运算，就变成“11111101”。

5.1.3 认识MM74C922/MM74C923

键盘状态的检测，除了可以利用键盘扫描软件外，还可应用现成的键盘扫描IC，例如NS半导体公司所提供的MM74C922及MM74C923为键盘扫描IC，其中MM74C922为4×4的键盘扫描IC、MM74C923为4×5的键盘扫描IC，如图5-10所示，对其中各引脚说明如下。

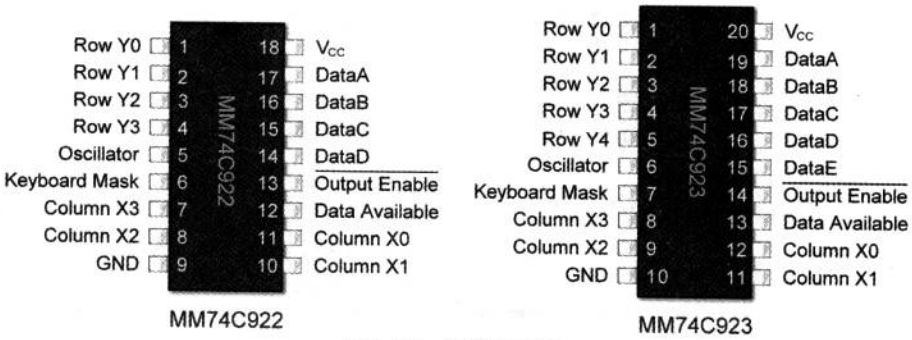
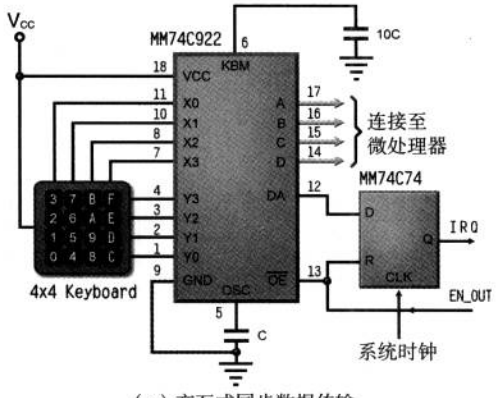
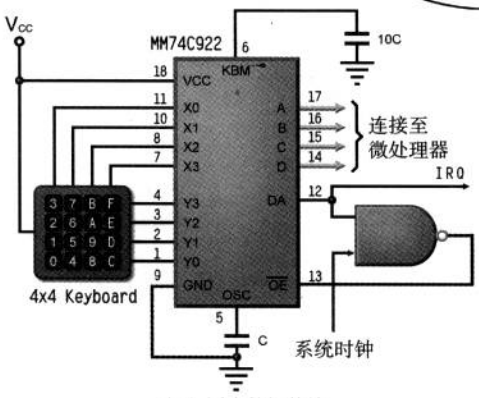


图5-10 键盘扫描IC

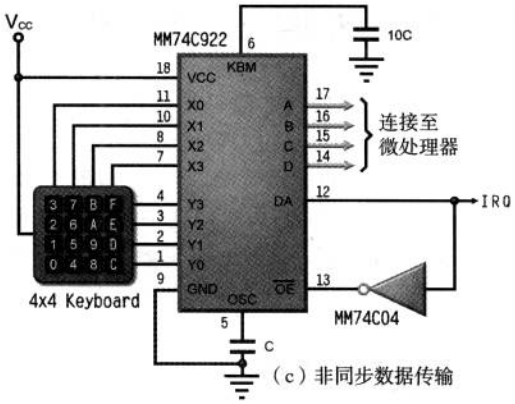
- **DataA~DataE**：输出检测键盘的结果，连接到微处理器的输入端口。若是MM74C922，则只有DataA~DataD4只引脚。
- **ColumnX0~ColumnX3**：连接键盘的X0~X3行。
- **RowY0~RowY4**：连接键盘的Y0~Y4列，若是MM74C922，则只有RowY0~RowY34只引脚。
- **Oscillator**：本引脚为振荡，简称OSC引脚，本引脚必须连接电容，让MM74C922（MM74C923）内部产生脉冲，通常连接0.1μF的电容即可。
- **Keyboard Mask**：本引脚为按键遮没引脚，简称为KBM引脚，其功能是提供遮盖按键抖动的周期，也就是一种硬件防抖动。该引脚需要连接一个电容C，其内部有一个10kΩ电阻R，它形成一个充放电周期（0.7RC），即为遮没周期，电容值大小可为OSC引脚所连接电容的10倍，约为1μF。当按键按下时，即进入遮没周期，首先将暂停IC内部的计数，同时DA引脚变为高电平，一直到按键被放开，DA引脚才恢复为低电平。
- **Data Available**：本引脚为允许数据输出引脚，简称为DA引脚。平时该引脚为低电平，此IC的数据输出引脚DataA~DataE不提供正确的键盘状态数据。当本引脚为高电平时，则键盘状态数据将由数据输出引脚DataA~DataE输出。
- **Output Enable**：本引脚为输出启用引脚，简称为OE引脚，属于低电平驱动引脚。我们可通过简单的逻辑闸或正反器，以规划本IC的数据输出引脚DataA~DataE与微处理器之间的传输方式。而数据传输方式可为同步交互式数据传输、同步数据传输，及异步数据传输等，如图5-11所示。



（a）交互式同步数据传输



（b）同步数据传输



（c）非同步数据传输

图5-11 数据传输模式

MM74C922/MM74C923对各按键的响应如表5-2所示。

表5-2

按键	连接引脚	DataE	DataD	DataC	DataB	DataA
0	X0/Y0	0	0	0	0	0
1	X1/Y0	0	0	0	0	1
2	X2/Y0	0	0	0	1	0
3	X3/Y0	0	0	0	1	1
4	X0/Y1	0	0	1	0	0
5	X1/Y1	0	0	1	0	1
6	X2/Y1	0	0	1	1	0
7	X3/Y1	0	0	1	1	1
8	X0/Y2	0	1	0	0	0
9	X1/Y2	0	1	0	0	1
10	X2/Y2	0	1	0	1	0
11	X3/Y2	0	1	0	1	1

续表

按键	连接引脚	DataE	DataD	DataC	DataB	DataA
12	X0/Y3	0	1	1	0	0
13	X1/Y3	0	1	1	0	1
14	X2/Y3	0	1	1	1	0
15	X3/Y3	0	1	1	1	1
16	X0/Y4	1	0	0	0	0
17	X1/Y4	1	0	0	0	1
18	X2/Y4	1	0	0	1	0
19	X3/Y4	1	0	0	1	1

5.2 7段LED数码管扫描



在第3章中，我们已介绍过单个7段LED数码管及其应用。同样地，若要同时使用多位数7段LED数码管时，若还是与单个7段LED数码管一样，采用分别（独立）驱动方式的话，效率就会很低！当然，采用分别驱动每个7段LED数码管的方式，也将耗用较多的器件与成本。在此将介绍将多个7段LED数码管包在一起的7段LED数码管模块，以及利用人类的视觉暂留现象，快速扫描的驱动方式，则只要一组驱动电路即可达到显示多个7段LED数码管的目的。

5.2.1 认识7段LED数码管模块

在此要介绍由数个单位数7段LED数码管所组成的多位数7段LED数码管以及使用多位数7段LED数码管模块的应用。

● 多个7段LED数码管组合

若要同时使用多个7段LED数码管，必须采用扫描式显示。在硬件电路方面，首先将每个7段LED数码管的a、b…g都连接在一起，再使用晶体管分别驱动每个7段LED数码管的共同引脚com。以4个共阳极7段LED数码管为例，按如图5-12所示连接电路。

其显示方式是将第一个7段LED数码管所要显示的数据送到a、b…g总线上，然后将1110扫描信号送到4个晶体管的基极，即可显示第一个7段LED数码管；若要显示第二个7段LED数码管，同样是将所要显示的数据丢到a、b…g总线上，然后将1101扫描信号送到4个晶体管的基极，即可显示第二个7段LED数码管；若要显示第3个7段LED数码管，同样是将所要显示的数据送到a、b…g总线上，然后将1011扫描信号送到4个晶体管的基极，即可显示第3个7段LED数码管；若要显示第4个7段LED数码管，同样是将所要显示的数据送到a、b…g总线上，然后将0111扫描信号送到4个晶体管的基极，即可显示第4个7段LED数码管。扫描一圈后，再从头开始扫描。虽然任何一个时间里只有一个7段LED数码管在显示，但只要从第一个到最后一个的扫描时间不超过16ms，人眼就会同时看到这几个数字。

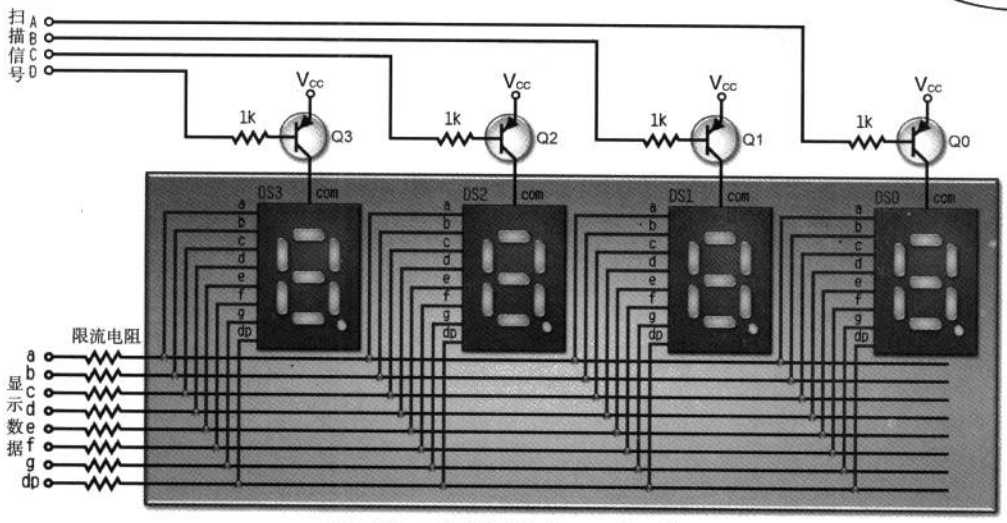


图5-12 4个共阳极7段LED数码管

由此可知，以扫描方式驱动多个并接的7段LED数码管时，驱动信号包括显示数据与扫描信号，“显示数据”是所要显示的驱动信号编码，与驱动单位数7段LED数码管一样（详见3.2.4节）；“扫描信号”就像是开关，用来决定驱动哪一个位数。而“扫描信号”也分成高电平扫描与低电平扫描两种，采用哪种扫描信号与电路结构有关。以图5-12为例，其扫描信号分别接入Q0~Q3的PNP晶体管的基极，其中低电平者将使其所连接的晶体管导通，所驱动的那一位才可能会显示，称之为低电平扫描。若把Q0~Q3改为NPN晶体管，而且E、C对调，则需要高电平信号才能使晶体管导通，称之为高电平扫描。采用低电平扫描比较常见。

7段LED数码管模块

7段LED数码管模块是把多个位数的7段LED数码管包在一起，其中各位数的a都连接到a引脚、各位数的b都连接到b引脚、各位数的c都连接到c引脚……，而每个位数的共同引脚是独立的。市面上常见的7段LED数码管模块有两位数、三位数、四位数、六位数等。下面以四位数7段LED数码管模块为例进行讲解，实物如图5-13所示。

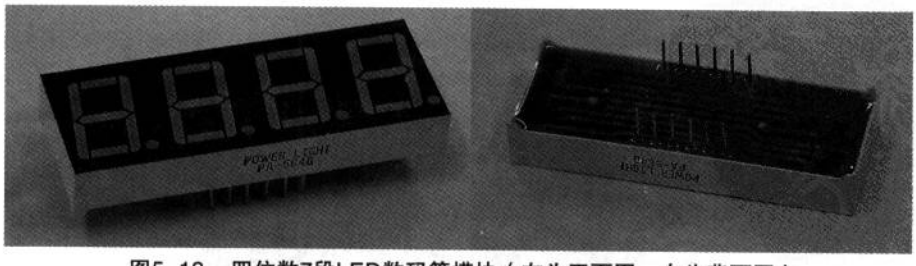


图5-13 四位数7段LED数码管模块（左为正面图、右为背面图）

四位数7段LED数码管模块便宜又好用（约4个单位数7段LED数码管的一半价钱），它的尺寸与内部结构图如图5-14所示。

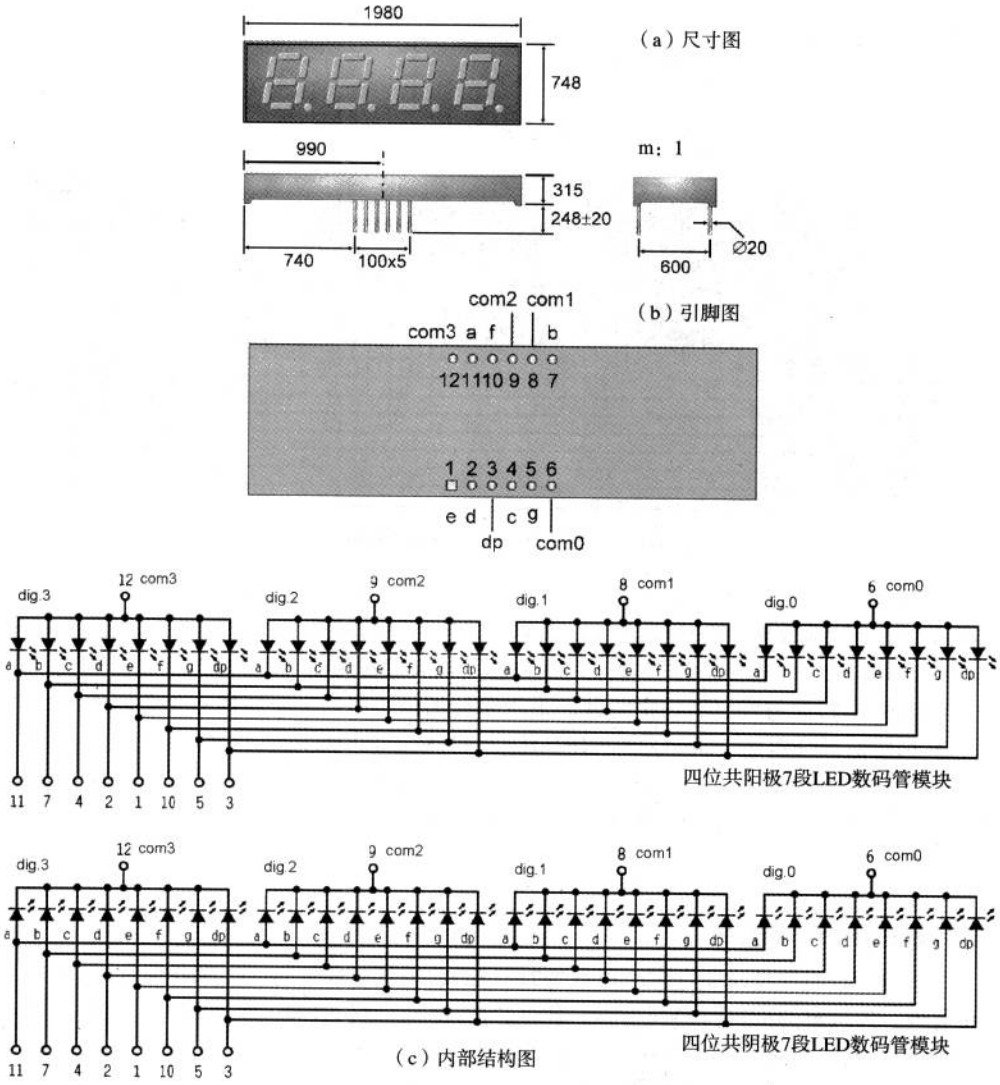
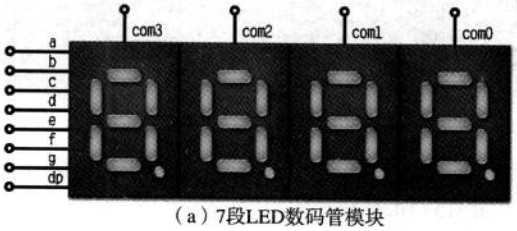
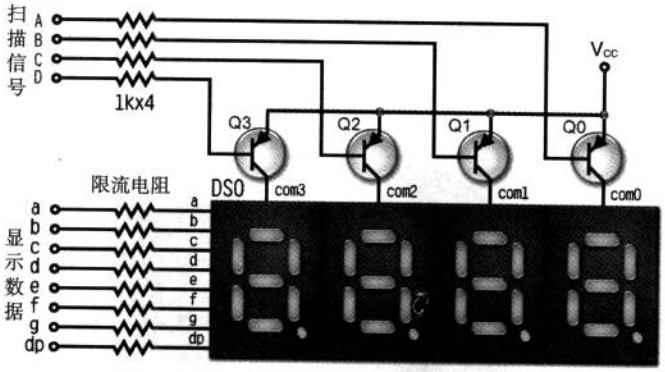


图5-14　四位数7段LED数码管模块

很明显，其中只有a、b…g、dp以及com0~com3等12个引脚，简单多了！以图5-12中的电路为例，当我们使用4个7段LED数码管时，每个7段LED数码管都有a、b…g，线路复杂的很！若改用四位数7段LED数码管模块，就非常简单，如图5-15所示。



（a）7段LED数码管模块



（b）7段LED数码管的应用

图5-15 7段LED数码管模块

● 扫描驱动的问题

对于扫描方式驱动的LED或7段LED数码管，其亮度与稳定度是个问题。如果要亮一点，则扫描的频率要低一点，以提高工作周期；但是如果扫描频率太低，则会有闪烁的感觉。因此，我们还是建议把扫描频率限制在60Hz以上，也就是在16ms之内完整扫描一周，才不会闪烁。若为四位数（或四组负载）的扫描，其每位数（或四组负载）的工作周期为固定式负载的1/4，其亮度约为固定式负载的1/4；若是八位数（或八组负载）的扫描，那工作周期只剩下固定式负载的1/8，那么亮度就更小！如何提高亮度呢？在此有如下两个建议。

▶ 降低限流电阻值。在第3章里曾经介绍过，若要驱动一个LED，或一位数的7段LED数码管，除了驱动信号（5V）外，还必须串接限流电阻。限流电阻的电阻值在200~330Ω，让其顺向电流限制在10~20mA。对于扫描方式驱动的LED或7段LED数码管，需要再降低限流电阻的值。

- 四位数的扫描：可使用75~150Ω的限流电阻，其电流将限制在22~44mA。若整个扫描周期为16ms，每位数约4ms点亮。当然，并不会一下就点亮，需要有响应时间，电流的上升也需要时间。所以，在工作周期内，电流并没有计算得那么大。
- 八位数的扫描：可使用50~75Ω的限流电阻，其电流将限制在44~66mA。若整个扫描周期为16ms，每位数约2ms点亮。当然，并不会一下就点亮，需要有响应时间，电流的上升也需要时间。所以，在工作周期内，电流并没有计算的那么大。

上述降低限流电阻的方法，在进行在线仿真时要小心！若程序停止或暂停时，LED可能持续点亮。这时候，可能就会有22~44mA（四位数）或44~66mA（八位数）电流流过LED，即使不会马上破坏该LED，也会降低其寿命。

▶ 选用高亮度的7段LED数码管模块。LED技术发展很快，市面上不乏高亮度的产品。当然，高亮度的LED或7段LED数码管的驱动电流与顺向电压不见得与此所介绍的相同，所以要记得索取或参考其使用说时作为设计的依据。

5.2.2 认识7447/7448

BCD码转换成7段LED数码管的译码驱动IC，首推7447系列，包括7446、7447、

7448和74LS49。其中的7446及7447输出低电平驱动的显示码，用以驱动共阳极7段LED数码管；而7448及74LS49输出高电平驱动的显示码用以驱动共阴极7段LED数码管。7446、7447与7448的引脚相同（双并排16Pins），74LS49则为双并排14Pins，如图5-16所示：

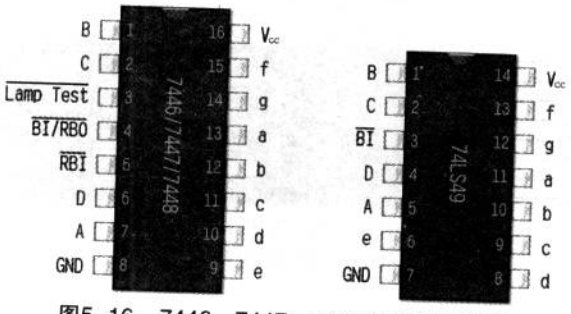


图5-16 7446、7447、7448和7449的引脚

引脚说明

- **D**、**C**、**B**、**A**：BCD码输入引脚。
- **a**、**b**、**c**…**g**：7段LED数码管输出引脚。
- $\overline{\text{LampTest}}$：本引脚为测试引脚，简称为 $\overline{\text{LT}}$引脚。当本引脚输入低电平时，所连接的7段LED数码管将全亮。正常显示下，本引脚应输入高电平。
- $\overline{\text{RBI}}$：本引脚为涟波遮没输入引脚（ripple-blanking input）。正常显示下，本引脚应输入高电平。若本引脚输入低电平（即0），且D、C、B、A引脚输入为0，则该位数将不显示，这项功能称为消除前置0（leading zero suppression）或消除尾端0（trailing zero suppression），稍后将具体说明。
- $\overline{\text{BI/RBO}}$：本引脚为遮没输入或涟波遮没输出引脚（blanking in and/or ripple-blanking output）。正常显示下，本引脚应输入高电平或悬空。若本引脚连接低电平（0），则该位数将不显示。当该位数不显示时，本引脚将输出低电平，以串接到前一个位数的RBI引脚，作为消除前置0（leading zero suppression）或消除尾端0（trailing zero suppression）之用，稍后将具体说明。

真值表

表5-3所示为7446、7447和7448的真值表，表5-4所示为74LS49的真值表。

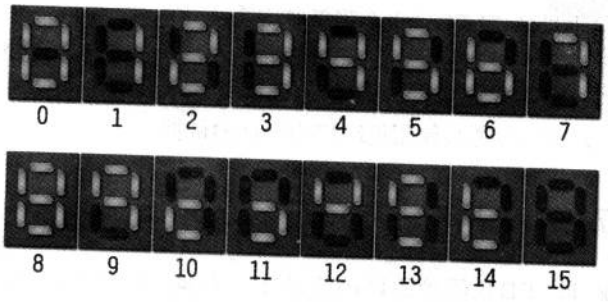


表5-3

数字或功能	输入						BI/RBO	7446/7447 输出							7448 输出						
	$\overline{LT}$	$\overline{RBI}$	D	C	B	A		a	b	c	d	e	f	g	a	b	c	d	e	f	g
0	1	1	0	0	0	0	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0
1	1	X	0	0	0	1	1	1	0	0	1	1	1	1	0	1	1	0	0	0	0
2	1	X	0	0	1	0	1	0	0	1	0	0	1	0	1	1	0	1	1	0	1
3	1	X	0	0	1	1	1	0	0	0	0	1	1	0	1	1	1	1	0	0	1
4	1	X	0	1	0	0	1	1	0	0	1	1	0	0	0	1	1	0	0	1	1
5	1	X	0	1	0	1	1	0	1	0	0	1	0	0	1	0	1	1	0	1	1
6	1	X	0	1	1	0	1	1	1	0	0	0	0	0	0	0	1	1	1	1	1
7	1	X	0	1	1	1	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0
8	1	X	1	0	0	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1
9	1	X	1	0	0	1	1	0	0	0	1	1	0	0	1	1	1	0	0	1	1
10	1	X	1	0	1	0	1	1	1	1	0	0	1	0	0	0	0	1	1	0	1
11	1	X	1	0	1	1	1	1	1	0	0	1	1	0	0	0	1	1	0	0	1
12	1	X	1	1	0	0	1	1	0	1	1	1	0	0	0	1	0	0	0	1	1
13	1	X	1	1	0	1	1	0	1	1	0	1	0	0	1	0	0	1	0	1	1
14	1	X	1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	1	1	1	1
15	1	X	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
BI	X	X	X	X	X	X	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
RBI	1	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0
LT	0	X	X	X	X	X	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1

X：表示可为0或1

表5-4

数字或功能	输入					输出						
	D	C	B	A	BI	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	1	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	1	0	0	1
4	0	1	0	0	1	0	1	1	0	0	1	1
5	0	1	0	1	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	0	1	1	1	1	1
7	0	1	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	0	1	1
10	1	0	1	0	1	0	0	0	1	1	0	1
11	1	0	1	1	1	0	0	1	1	0	0	1

续表

数字或功能	输入					输出						
	D	C	B	A	BI	a	b	c	d	e	f	g
12	1	1	0	0	1	0	1	0	0	0	1	1
13	1	1	0	1	1	1	0	0	1	0	1	1
14	1	1	1	0	1	0	0	0	1	1	1	1
15	1	1	1	1	1	0	0	0	0	0	0	0
BI	X	X	X	X	0	0	0	0	0	0	0	0

X：表示可为0或1

消除前置0

所谓“消除前置0”是指在数字整数部份的左边位数若为0，则不显示该位数，例如“012”则只显示“12”、“002”则只显示“2”。若要使用7446、7447和7448所提供的消除前置0功能，则可将整数部分最左边位数的RBI引脚接地，BI/RBO引脚连接到其右边位数的RBI引脚……，依次类推。不过，个位数的RBI引脚并不与前左边的BI/RBO引脚连接，以避免全部整数都不显示；同时，将其dp引脚连接限流电阻后接地，以显示小数点。如图5-17所示，其中整数部分输入“0000 0001 0001”，则7段LED数码管显示“11.”。

消除尾端0

所谓“消除尾端0”是指在数字小数部分的右边位数若为0，则不显示该位数，例如“0.150”则只显示“0.15”、“0.200”则只显示“0.2”。若要使用7446、7447和7448所提供的消除尾端0功能，则可将小数部分最右边位数的RBI引脚接地、BI/RBO引脚连接到其左边位数的RBI引脚……，依次类推。如图5-17所示，其中整数部分输入“0101 0000”，则7段LED数码管显示“.5”。

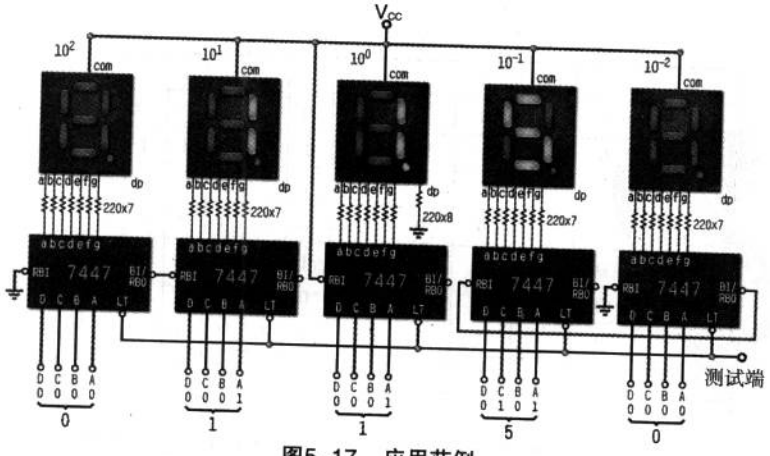


图5-17 应用范例

测试7段LED数码管

若将LT引脚接地，则7段LED数码管将全亮，通过这种方法可以检查是否有不显示的节。对于多个7段LED数码管的电路（如图5-17所示），则可将每个LT引脚连接起来作为测试端，平时将此测试端连接高电平，若要测试7段LED数码管，则将它接地即可。

这4个IC都是集电极式输出，所以其输出电流足以驱动7段LED数码管，其中7448的输出端已内含2kΩ的上拉电阻，其他3个IC的输出端都没有内含上拉电阻。除了低电平输出与高电平输出的不同外，对于集电极式输出所连接的负载，其所连接的电源电压也有些差异，如表5-5所示。

表5-5

	最大负载电压	最大流入电流
7446	30V	40 mA
7447	15V	40 mA
7448	5.5V	6.4 mA
74LS49	5.5V	8 mA

5.2.3 认识74138/74139

不管是键盘扫描，还是多个7段LED数码管的扫描，扫描信号都是不可或缺的！当然，我们可以在微处理器里用软件产生扫描信号，对于8051，若第一个扫描信号为“11111110”，则可利用“<<1”左移指令产生下一个扫描信号。不过，如果直接由微处理器输出扫描信号，必须使用较多的输出口。我们可以利用“2-4线译码器”或“3-8线译码器”来产生4位或8位的扫描信号。74139内含两个2-4线译码器，而74138则为3-8线译码器，如图5-18所示。

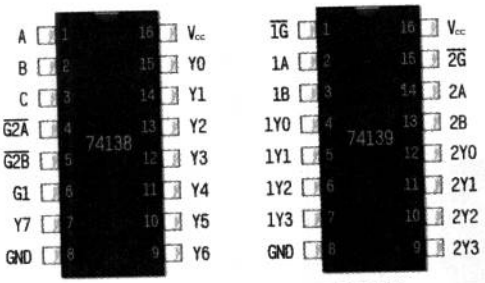


图5-18 74139、74138的引脚

74138引脚说明

- **C、B、A**：2进制码输入引脚。
- **Y7、Y6、Y5…Y0**：扫描信号输出引脚。

- $\overline{G2A}$、$\overline{G2B}$：低电平启用引脚，若$\overline{G2A}$或$\overline{G2B}$为1，则此译码器不工作，输出引脚（Y7、Y6、Y5…Y0）将全部输出为1。若$\overline{G2A}$与$\overline{G2B}$都为0，则此译码器才可能正常工作。
- G1：高电平启用引脚，若本引脚为0，则此译码器不工作，输出引脚（Y7、Y6、Y5…Y0）将全部输出为1。若本引脚为1，则此译码器才可能正常工作。

74139引脚说明

- **1B**、**1A**：第一个译码器的BCD码输入引脚。
- **1Y3**、**1Y2**、**1Y1**、**1Y0**：第一个译码器的扫描信号输出引脚。
- $\overline{1G}$：第一个译码器的低电平启用引脚，若$\overline{1G}$为1，则第一个译码器不工作，输出引脚（1Y3、1Y2、1Y1、1Y0）将全部输出为1。若$\overline{1G}$为0，则第一个译码器将正常工作。
- **2B**、**2A**：第二个译码器的BCD码输入引脚。
- **2Y3**、**2Y2**、**2Y1**、**2Y0**：第二个译码器的扫描信号输出引脚。
- $\overline{2G}$：第二个译码器的低电平启用引脚，若$\overline{2G}$为1，则第二个译码器不工作，输出引脚（2Y3、2Y2、2Y1、2Y0）将全部输出为1。若$\overline{2G}$为0，则第二个译码器将正常工作。

真值表

74138的真值表如表5-6所示，74139的真值表如表5-7所示。

表5-6

输入						输出							
启用			资料										
G1	$\overline{G2A}$	$\overline{G2B}$	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	1	1	X	X	X	1	1	1	1	1	1	1	1
X	0	1	X	X	X	1	1	1	1	1	1	1	1
X	1	0	X	X	X	1	1	1	1	1	1	1	1
0	X	X	X	X	X	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	1	1	1	1	1	1
1	0	0	0	0	1	1	0	1	1	1	1	1	1
1	0	0	0	1	0	1	1	0	1	1	1	1	1
1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	0	1	1	1	1	0	1	1	1
1	0	0	1	0	1	1	1	1	1	1	0	1	1
1	0	0	1	1	0	1	1	1	1	1	1	0	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0

X：表示可为0或1

表5-7

输 入			输 出			
$\overline{1G}$	1B	1A	1Y0	1Y1	1Y2	1Y3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

输 入			输 出			
$\overline{2G}$	2B	2A	2Y0	2Y1	2Y2	2Y3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

X：表示可为0或1

5.3 静态显示与动态显示



下面将介绍7段LED数码管模块的扫描与动态显示，其中的动态显示包括闪烁、交替显示、一个字一个字飞入显示器以及常见的跑马灯方式等。

5.3.1 使用BCD译码器

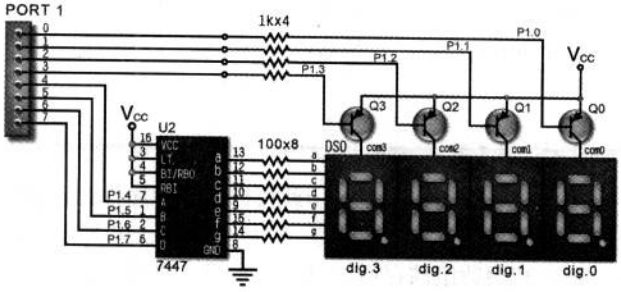


图5-19 使用7447译码器的7段LED数码管模块驱动电路图之一

如图5-19所示，在这个驱动电路里，扫描信号通过P1.3~P1.0送至Q3~Q0的基极，以低电平驱动7段LED数码管模块的digit 3~digit 0。而显示信号（BCD码）将通过P1.7~P1.4送至7447译码，7447译码的结果再接到7段LED数码管模块的a~g。这里7447的LT、BI/RBO及RBI引脚要连接到V_{cc}，其实这3个脚悬空也没关系，对于TTL，输入引脚悬空相当于输入High。但这样比较易受噪声干扰，所以大家要养成好习惯，不用的输入引脚尽量不要悬空。

整个扫描分为4个阶段，以显示“8051”为例说明如下。

- 第1阶段要在digit 0显示1，所以显示数据为1，也就是P1.7~P1.4的内容为1；扫描信号为1110（即16进制的e），通过P1.3~P1.0送出。因此，我们只要将“0x1e”送到Port 1，7段LED数码管模块的digit 0将显示“1”。延迟4ms后，再进入第2阶段。
- 第2阶段要在digit 1显示5，所以显示数据为5，扫描信号为1101（即16进制的d）。因此，我们只要将“0x5d”送到Port 1，7段LED数码管模块的digit 1将显示“5”。延迟4ms后，再进入第3阶段。

- 第3阶段要在digit 2显示0，所以显示数据为0、扫描信号为1011（即16进制的b）。因此，我们只要将“0x0b”送到Port 1，7段LED数码管模块的digit 2将显示“0”。延迟4ms后，再进入第4阶段。
- 第4阶段要在digit 3显示8，所以显示数据为8、扫描信号为0111（即16进制的7）。因此，我们只要将“0x87”送到Port 1，7段LED数码管模块的digit 1将显示“8”。延迟4ms后，再从第一阶段开始扫描。

在程序设计上，可利用“土法炼钢”的方式，把驱动信号一个接一个地送到Port 1，使用如下指令。

```
main()
{    while(1)
     {   P1=0x1e;              // digit 0显示1
         delay1m(4);           // 延迟4ms
         P1=0x5d;              // digit 1显示
         delay1m(4) ) ;        // 延迟4ms
         P1=0x0b;              // digit 2显示0
         delay1m(4);           // 延迟4ms
         P1=0x87;              // digit 3显示8
         delay1m(4);           // 延迟4ms
     }                         // 结束while语句
}                              // 主程序结束
```

这种设计方式的优点是只要使用一个Port就可以了，程序设计也很简单。不过，若要改变显示内容（数字）就得从程序中慢慢找，扩展性较差。若把显示信号与驱动信号分开由两个Port输出，则可利用数组储存所要显示的数据，以移位的方式产生扫描信号，看起来就比较有水平。按图5-20所示连接电路。

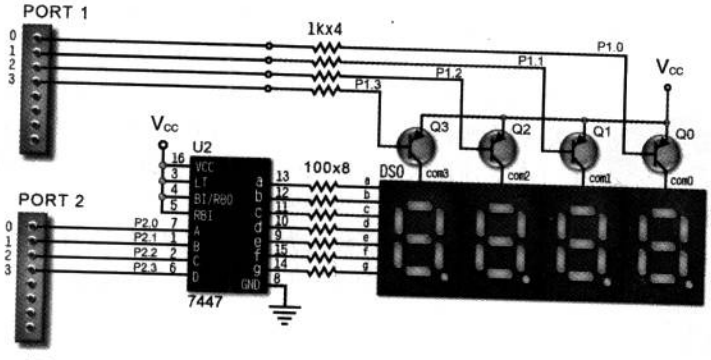


图5-20　使用7447译码器的7段LED数码管模块驱动电路图之二

对应的驱动程序如下。

```
char disp[4]={ 8, 0, 5, 1 };          // 声明显示数据
unsigned char scan;                   // 声明扫描信号
char i;                               // 声明变量i
main()
{    while(1)                         // while循环开始
     {   scan=1;                      // 设定扫描信号初始值
         for(i=3; i>=0; i--)          // for语句开始
         {  P2=disp[i];               // 输出显示信号
            P1=~scan;                 // 输出扫描信号
            delay1m(4);               // 延迟4ms
            scan<<=1;                 // 下一个扫描信号
         }                            // 结束for语句
     }                                // 结束while语句
}                                     // 主程序结束
```

或许从上面的例子中只看到程序设计比较有水平一点而已。不过，如果将显示的位数扩展时，就可看出更多的优点，图5-21所示为8位数的电路，而只要把上面的程序稍微改一下就可以了，如图5-21所示。

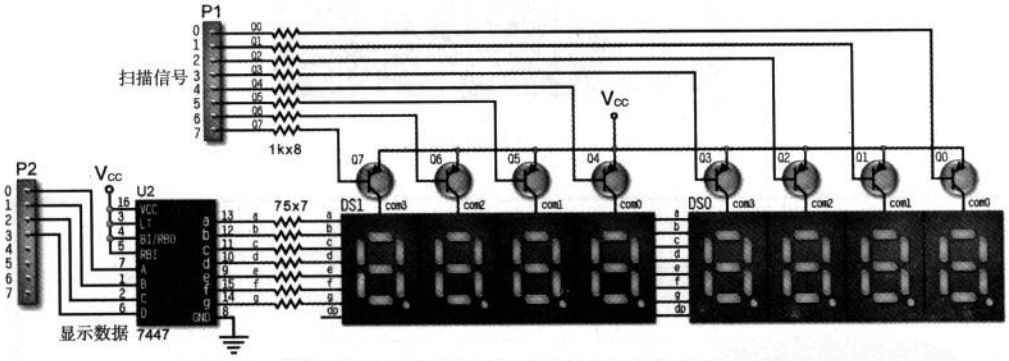


图5-21 使用7447译码器的7段LED数码管模块驱动电路图之三

对应的驱动程序如下。

```
char disp[8]={ 2, 0, 0, 8, 0, 3, 1, 5 };      // 声明显示数据
unsigned char scan;                           // 声明扫描信号
char i;                                       // 声明变量i
main()
{    while(1)                                 // while循环开始
     {   scan=1;                              // 设定扫描信号初始值
```

```
        for(i=7; i>=0; i--)                    // for语句开始
        {   P2=disp[i];                        // 输出显示信号
            P1=~scan;                          // 输出扫描信号
            delay1m(2);                        // 延迟2ms
            scan<<=1;                          // 下一个扫描信号
        }                                      // 结束for语句
    }                                          // 结束while语句
}                                              // 主程序结束
```

5.3.2 直接驱动

直接驱动方式如图5-22所示，直接以Port 1输出扫描信号，直接以Port 2输出显示驱动信号，不使用任何译码IC，以简化电路。而在程序设计上，也不过多出译码的程序而已，对应驱动程序如下。

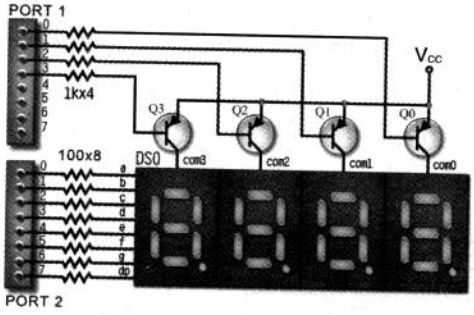


图5-22 直接驱动电路图

```
/*声明驱动信号数组*/
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,
               0x92, 0x83, 0xf8, 0x80, 0x98 };
//=========================================
char disp[4]={ 8, 0, 5, 1 };              // 声明显示数据
unsigned char scan;                       // 声明扫描信号
char i, j;                                // 声明变量i, j
main()
{   while(1)                              // while循环开始
    {   scan=1;                           // 设定扫描信号初始值
        for(i=3; i>=0; i--)               // for语句开始
        {   j=disp[i];                    // 取出显示数字
            P2=TAB[j];                    // 转换成驱动信号，并输出到P2
```

```
            P1=~scan;                    // 输出扫描信号
            delay1m(4);                  // 延迟4ms
            scan<<=1;                    // 下一个扫描信号
        }                                // 结束for语句
    }                                    // 结束while语句
}                                        // 主程序结束
```

由上面的程序可知，只做进行简单修改，即可省掉一些器件，当然，也降低了成本。所以，只要输入/输出端口够用，直接驱动是个好方法！

5.3.3 使用扫描译码器

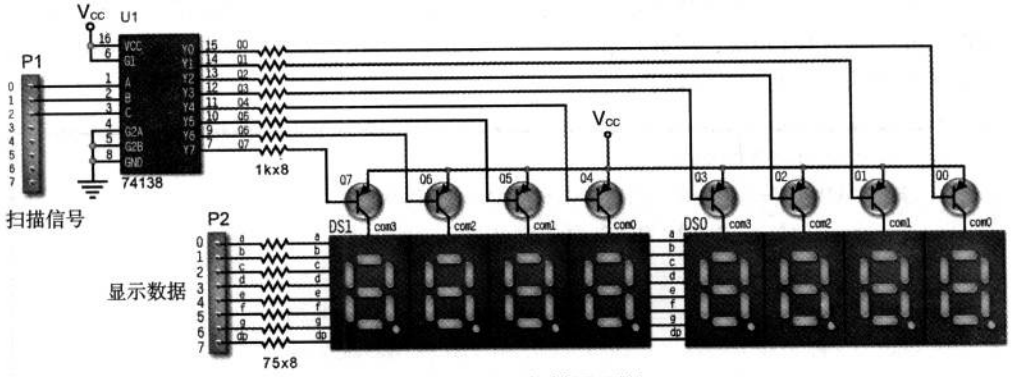


图5-23 使用扫描译码器

使用扫描译码器的方式如图5-23所示，Port 1输出BCD码，再经过74138产生扫描信号。使用74138要把G1引脚连接V_{CC}、G2A及G2B引脚接地；而Port 2输出显示驱动信号，不使用任何译码IC。使用74138产生扫描信号时，程序设计上，就不必刻意处理扫描信号，驱动程序如下。

```
/*声明驱动信号数组*/
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,
               0x92, 0x83, 0xf8, 0x80, 0x98 };
//=========================================
char disp[8]={ 2, 0, 0, 8, 0, 3, 1, 5 };     // 声明显示数据
char i, j;                                   // 声明变量i, j
main()
{   while(1)                                 // while循环开始
    {   for(i=0; i<8; i++)                   // for语句开始
        {   j=disp[7-i];                     // 取出显示数字
```

```
            P2=TAB[j];                    //转换成驱动信号，并输出到P2
            P1=i;                         //输出扫描信号
            delay1m(2);                   //延迟2ms
        }                                 //结束for语句
    }                                     //结束while语句
}                                         //主程序结束
```

5.3.4 闪烁

所谓“闪烁”就是时亮时不亮的效果，以直接驱动四位数7段LED数码管模块（如图5-22所示）为例，若要显示“8051”，扫描一次花16ms。如果希望这4个数字显示约0.48s，然后不显示0.48s，如此交替循环，所呈现出来的就是这4个数字“亮-暗”闪烁。0.48s（即480ms）可扫描30次，因此，我们只要持续执行“30次的扫描工作，再关闭0.48s”，即可达到闪烁的效果，程序如下。

```
/*声明驱动信号数组*/
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,
               0x92, 0x83, 0xf8, 0x80, 0x98 };
//=========================================
char disp[4]={ 8, 0, 5, 1 };              //声明显示数据
char scan;                                //声明扫描信号
char i, j, k;                             //声明变量i, j, k
main()
{   while(1)                              //while循环开始
    {  for(k=0; k<30; k++)                //设定30次扫描
       {  scan=1;                         //设定扫描信号初始值
          for(i=3; i<0; i--)              //for语句开始
          {  j=disp[i];                   //取出显示数字
             P2=TAB[j];                   //转换成驱动信号，并输出到P2
             P1=~scan;                    //输出扫描信号
             delay1m(4);                  //延迟4ms
             scan<<=1;                    //下一个扫描信号
          }                               //结束i循环
       }                                  //结束k循环
       P2=0xff;                           //全暗
       delay1m(480);                      //延迟0.48s
    }                                     //结束while语句
}                                         //主程序结束
```

5.3.5 交替显示

所谓“交替显示”就是多组数字切换显示，以直接驱动四位数7段LED数码管模块（如图5-22所示）为例，若要显示“2008”，然后显示“0315”0.48s，如此交替循环。则只要持续执行“30次扫描‘2008’，再30次扫描‘0315’”，即可达到交替显示的效果，程序如下。

```
/*声明驱动信号数组*/
char TAB[10]={  0xc0, 0xf9, 0xa4, 0xb0, 0x99,
                0x92, 0x83, 0xf8, 0x80, 0x98 };
//=========================================
char disp1[4]={ 2, 0, 0, 8 };    // 声明显示数据1
char disp2[4]={ 0, 3, 1, 5 };    // 声明显示数据2
char scan;                       // 声明扫描信号
char i, j, k;                    // 声明变量i, j, k
main()
{    while(1)                    // while循环开始
     {    for(k=0; k<30; k++)    // 设定30次扫描
          {   scan=1;            // 设定扫描信号初始值
              for(i=3; i<0; i--) // for语句开始
              {   j=disp1[i];    // 取出第一组显示数字
                  P2=TAB[j];     // 转换成驱动信号，并输出到P2
                  P1=~scan;      // 输出扫描信号
                  delay1m(4);    // 延迟4ms
                  scan<<=1;      // 下一个扫描信号
              }                  // 结束i循环
          }                      // 结束k循环
//=========================================
          for(k=0; k<30; k++)    // 设定30次扫描
          {   scan=1;            // 设定扫描信号初始值
              for(i=3; i<0; i--) // for语句开始
              {   j=disp2[i];    // 取出第二组显示数字
                  P2=TAB[j];     // 转换成驱动信号，并输出到P2
                  P1=~scan;      // 输出扫描信号
                  delay1m(4);    // 延迟4ms
                  scan<<=1;      // 下一个扫描信号
              }                  // 结束i循环
          }                      // 结束k循环
     }                           // 结束while语句
}                                // 主程序结束
```

5.3.6 飞入

相对于“闪烁”及“交替显示”，“飞入”的动作就比较复杂，例如要将“8051”4个数字由右边飞入7段LED数码管模块，其动作可分为以下10组扫描动作。

其中空白部分为不亮，其驱动信号编码为0xff，在此将它放入TAB[10]位置。另外，我们可以把前面所介绍的扫描四位数程序写成一个函数，而所要显示的数字分成每4个一组，放入disp[10][4]数组中，其中第1组为“10，10，10，8”，第2组为“10，10，8，10”…，整个disp[10][4]数组的声明如下。

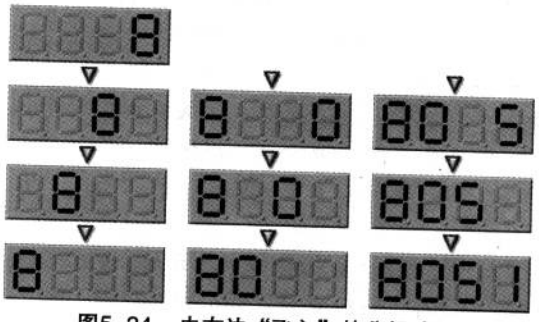


图5-24 由右边“飞入”的分解动作

```
char disp[10][4]={      {10, 10, 10, 8}, {10, 10, 8, 10},
                        {10, 8, 10, 10}, {8, 10, 10, 10},
                        {8, 10, 10, 0}, {8, 10, 0, 10},
                        {8, 0, 10, 10},
                        {8, 0, 10, 5}, {8, 0, 5, 10},
                        {8, 0, 5, 1}  };
```

整个程序如下。

```
/*声明驱动信号数组*/
char TAB[11]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,
               0x92, 0x83, 0xf8, 0x80, 0x98, 0xff };
/*声明显示数据数组*/
char disp[10][4]={  {10, 10, 10, 8}, {10, 10, 8, 10},
                    {10, 8, 10, 10}, {8, 10, 10, 10},
                    {8, 10, 10, 0}, {8, 10, 0, 10},
                    {8, 0, 10, 10},
                    {8, 0, 10, 5}, {8, 0, 5, 10},
                    {8, 0, 5, 1}      };
```

```
/*声明扫描函数*/
void scanner(char);
main()                                  //主程序开始
{    char   i;                          //声明i变量
     while(1)                           //while循环开始
     {   for(i=0; i<10; i++)            //for循环开始
         scanner(i);  }                 //扫描
}                                       //主程序结束
//=========================================
void scanner(char x)
{    char i, j, k;                      //声明变量i, j, k
     char scan;                         //声明扫描信号
     for(k=0; k<30; k++)                //设定30次扫描
     {   scan=1;                        //设定扫描信号初始值
         for(i=3; i<0; i--)             //for语句开始
         {   j=disp[x][i];              //取出第一组显示数字
             P2=TAB[j];                 //转换成驱动信号, 并输出到P2
             P1=~scan;                  //输出扫描信号
             delay1m(4);                //延迟4ms
             scan<<=1;                  //下一个扫描信号
         }                              //结束i循环
     }                                  //结束k循环
}                                       //结束scan函数
```

5.3.7 跑马灯

所谓“跑马灯”是指数字按顺序进入7段LED数码管模块，且连续不断，与“飞入”的动作有点像，只是显示的数据不同而已。例如要将“8051”4个数字由右边进入7段LED数码管模块，再由左边走出7段LED数码管模块，其动作可分为下列8组扫描动作，如图5-25所示。

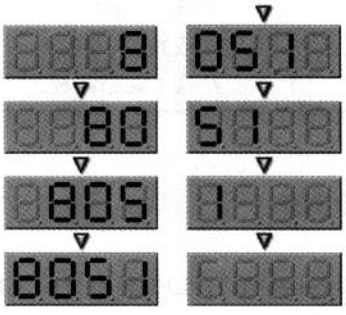


图5-25 由右边“跑入”的分解动作

整个程序如下。

```
/*声明驱动信号数组*/
char TAB[11]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,
               0x92, 0x83, 0xf8, 0x80, 0x98, 0xff };
/*声明显示数据数组*/
char disp[10][4]={  {10, 10, 10, 8}, {10, 10, 8, 0},
                    {10, 8, 0, 5}, {8, 0, 5, 1},
                    {0, 5, 1, 10}, {5, 1, 10, 10},
                    {1, 10, 10, 10}, {10, 10, 10, 10}     };
/*声明扫描函数*/
void scanner(char ) ;
main()                              //主程序开始
{   char    i;                      //声明i变量
    while(1)                        //while循环开始
    {  for(i=0; i<8; i++)           //for循环开始
       scanner(i); }                //扫描
}                                   //主程序结束
//=============================================
void scanner(char x)
{   char i, j, k;                   //声明变量i, j, k
    char scan;                      //声明扫描信号
    for(k=0; k<30; k++)             //设定30次扫描
    {   scan=1;                     //设定扫描信号初始值
        for(i=3; i<0; i--)          //for语句开始
        {   j=disp[x][i];           //取出第一组显示数字
            P2=TAB[j];              //转换成驱动信号，并输出到P2
            P1=~scan;               //输出扫描信号
            delay1m(4);             //延迟4ms
            scan<<=1;               //下一个扫描信号
        }                           //结束i循环
    }                               //结束k循环
}                                   //结束scan函数
```

5.4 同时扫描



如图5-26所示，4×4键盘与四位数7段LED数码管模块共享扫描信号。其中，键盘的

X0行与7段LED数码管模块的digit 0连接到P1.4，共享同一个扫描信号；键盘的X1行与7段LED数码管模块的digit 1连接到P1.5，共享同一个扫描信号；键盘的X2行与7段LED数码管模块0的digit 2连接到P1.6，共享同一个扫描信号；键盘的X3行与7段LED数码管模块的digit 3连接到P1.7，共享同一个扫描信号。不过，在看电路图时要格外小心，键盘的行编号是依据使用者的习惯由左向右编号的，左边开始为X0。7段LED数码管模块的位数是按数学的格式，最左边为MSD，最右边为LSD，所以最右边为digit 0。可以想象，扫描时键盘的扫描顺序是由左向右，7段LED数码管模块的扫描顺序是由右向左。当然，这并不会影响程序设计。

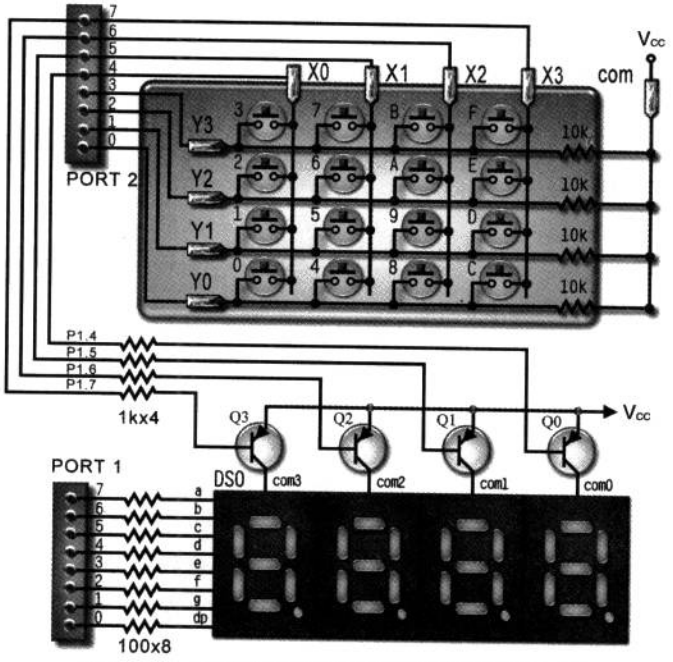


图5-26 4×4键盘与7段LED数码管同时扫描电路

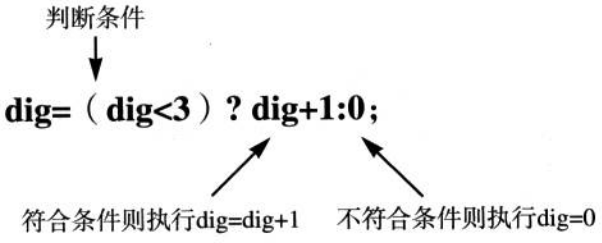
在此我们将利用for循环及if-else if语句来判断按键，其中也调用了debouncer函数来防抖动，但debouncer函数的内容没有列出来。一开始没有任何按下时，显示小数点；按下按键时，显示键值；若出现同时按下多个键等错误状态，显示负号。程序比较特别的是在此显示4个数字，所以需要用一个数组存放显示数据，整个程序如下。

```
#include<reg51.h>        // 定义8051寄存器的头文件
/* 声明7段LED数码管驱动信号数组(共阳) */
char TAB[18]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,        // 数字0-4
               0x92, 0x82, 0xf8, 0x80, 0x98,        // 数字5-9
               0xa0, 0x83, 0xa7, 0xa1, 0x84,        // 字母a-e(10-14)
               0x8e, 0xbf, 0x7f };          // 字母F(15), 负号(-), 小数点(.)
```

```
char disp[4]={ 0x7f, 0x7f, 0x7f, 0x7f };
// 声明4位7段LED数码管数组初值为小数点
char dig=0;                  // 声明7段LED数码管写入第几位(初值为最右边)
void debouncer(void);  // 声明防抖动函数
void scanner(void);        // 声明扫描函数
#define rowkey()   (~P1)&0x0f //读入P1低4位(列按键值)宏
//==主程序=====================================
main()                                    // 主程序开始
{     while(1)                            // 无穷循环，程序一直运行
          scanner();                      // 扫描函数
}                                         // 主程序结束
//==子程序=====================================
/* 防抖动函数，延迟约40ms */
void debouncer(void)                      // 防抖动函数开始
{     int i;                              // 声明整数变量i
      for(i=0; i<4800; i++);              // 计数4800次，延迟约40ms
}                                         // 防抖动函数结束
/* 扫描4×4键盘及4个7段LED数码管函数 */
void scanner(void)                        // 扫描函数开始
{     char col, row;                      // 声明变量(目前扫描第col行，第row列)
      char scan, keyin, kcode;            // 声明变量(扫描信号，列按键值，扫描码)
      scan=0xef;                          // 扫描信号初值=1110 1111
      for(col=0; col<4; col++ )           // for循环，扫描第col行
      {  P2=0xff;                         // 关闭7段LED数码管
         P1=scan;
         // P1高4位输出扫描信号，低4位设为输入
         P2=disp[col];                    // 输出第col行数字
         keyin=rowkey();                  // 读入P1低4位(第col行的列按键值)
         if(keyin != 0)                   // 若有按键时
         {  for(row=0; row<4; row++)      // 扫描第row列是否按下
            {  if(keyin == (0x01<<row))   // 只有第row列按下
               { kcode=row+4*col;         // 计算出按键码为
                 disp[dig]=TAB[kcode];    // 扫描码写入7段LED数码管数组
                 dig=(dig<3)? dig+1:0;    // 未满显示左1位，满时至最右位
                 break;                   // 离开第row列扫描(for循环)
               }                          // if语句(第row列按下)结束
              if (row==4)                 // 输入错误时
```

```
                { disp[dig]=TAB[16];          // 负号(-)写入7段LED数码管数组
                  dig=(dig<3)? dig+1:0;       // 未满显示左1位，满时至最右位
                }                             // if语句(输入错误)结束
              }                               // for循环，扫描row列结束
              while(rowkey() !=0);            // 等按钮放开
              debouncer();                    // 防抖动函数
            }                                 // if语句(有按键时)结束
            scan=(scan<<1)|0x01;              //产生左1位扫描信号
        }                                     // for循环，扫描col行结束
    }                                         // 扫描函数结束
```

- “disp[dig]=TAB[kcode]；”指令是将键值（按键状态）转换成显示码，并填入显示区数组disp[dig]，其中dig表示第几个位数。由于只有4个位数，所以使用“dig=（dig<3）? dig+1:0；”判断是否填满显示位数，基本上，这个指令是判断dig是否小于3，也就是0~2，如果小于3，则执行dig=dig+1；如果不小于3，也就是等于3，则执行dig=0，从第一个位数开始，这个有效率的指令格式如下：



5.5 实例演练



在本节中将提供键盘扫描、7段LED数码管扫描、译码器等的混合应用。

5.5.1 四位数7段LED数码管实例演练

● 实验要点

如图5-27所示，由P1高四位将所要显示的数字（BCD码）输出到7447，经7447译码后，驱动4位数字的7段LED数码管模块；而由P1低四位将扫描信号别分送到7段LED数码管模块的4个共同端，使这个7段LED数码管模块闪烁“2008”3次，再闪烁“0315”3次，如此循环不停。

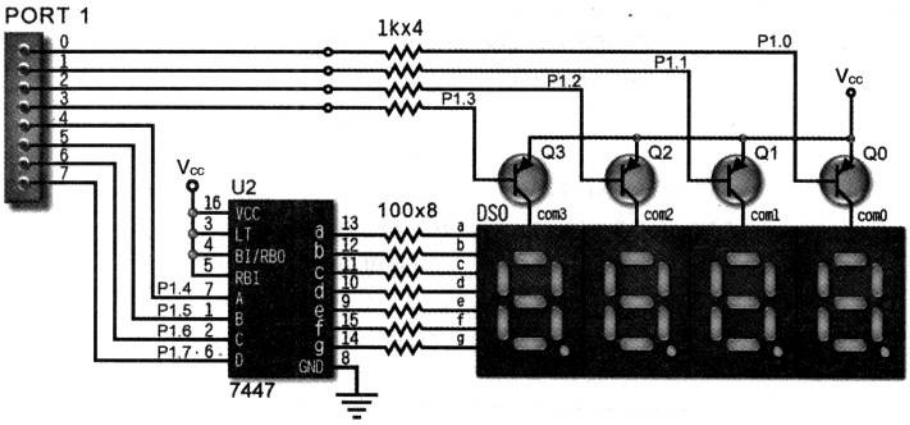


图5-27　四位7段LED数码管模阻电路图－使用7447

流程图与程序设计

基本上，在本实验里的程序是按5.3.1节中所介绍的方法设计的。比较难以理解的是如何读取数组里的数字数据，将扫描信号与显示数字数据结合。在此要将扫描码放在低四位，把所要显示的数字数据放在高四位，图5-28说明了如何读取数组数据。

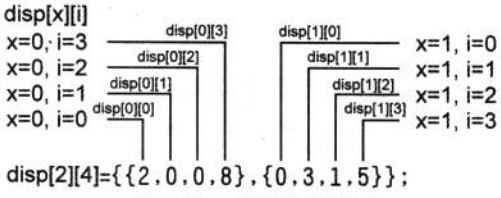


图5-28　读取数组数据

由于要从7段LED数码管模块最左边开始显示（扫描信号为0111），而这个二维数组里，disp[0][0]就是第一组中的第一个数据、disp[0][1]就是第一组中的第二个数据……，依次类推。所读取的数据放置在BCD变量里，而所要显示的数字是在其中的低四位，并不能马上输出到显示器。必须利用“BCD<<4”指令先把其中的低四位移至高四位。另外，将扫描信号scan与0x0f进行AND运算，使其高四位为0；再利用OR运算将其结果并入BCD，最后的结果为驱动此7段LED数码管模块的信号，整个数据处理过程如图5-29所示。

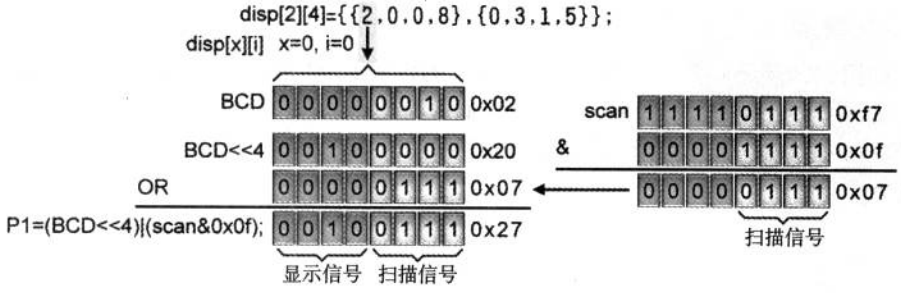
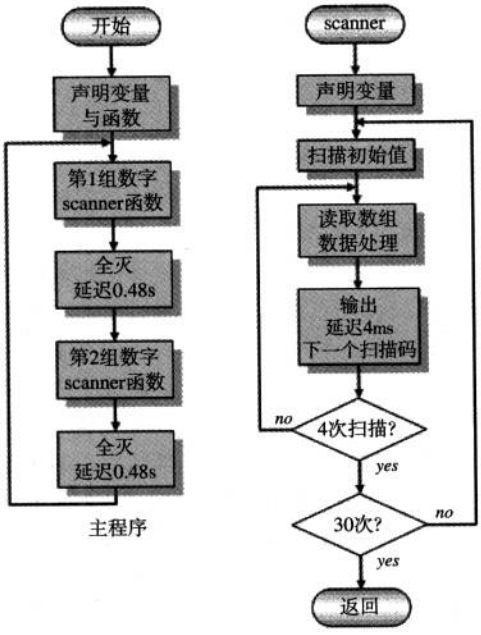


图5-29　数据处理




```
/* ch5_1.c - 7447解码的4位数7段LED数码管实验，P1.3~0为扫描信号，P1.7~4接7447 */
//==声明区==================================
#include<reg51.h>          // 定义8051寄存器的头文件，P2-17~19
char disp[2][4]={ {2，0，0，8},          // 显示数据(第0组)
                  {0，3，1，5}};         // 显示数据(第1组)
void delay1m(int);                       // 声明延迟函数
void scanner(char);                      // 扫描函数
//==主程序==================================
main()                                   // 主程序开始
{    char i;                             // 声明变量i
     while(1)                            // 无穷循环，程序一直运行
     {   for(i=0; i<2; i++)              // 显示第0，1列字组，for循环(字组i)开始
         {   scanner(i);                 // 扫描第i列字组
             P1=0xff;                    // 全灭
             delay1m(480);               // 延迟480×1m=0.48s
         }                               // for循环(字组i)结束
     }                                   // while循环结束
}                                        // 主程序结束
//==子程序==================================
/* 延迟函数，延迟约x×1ms */
```

```
void delay1m(int x)                          // 延迟函数开始
{    int i, j;                               // 声明整数变量i, j
     for (i=0; i<x; i++)                     // 计数x次，延迟x×1ms
          for (j=1; j<=120; j++);            // 计数120次，延迟1ms
}                                            // 延迟函数结束
/* 扫描字组函数，显示第x组数字 */
void scanner(char x)                         // 扫描字组函数开始
{    char i, j, BCD, scan;                   // 声明变量
     for (i=0; i<30; i++)                    // 扫描30次i循环
     {   scan=0xf7;                          // 扫描信号初值1111 0111
         for (j=0; j<4; j++)                 // 扫描4个数字j循环
         {   BCD=disp[x][j];                 // 读取第x组第j个数字的BCD码
             P1=(BCD<<4)|(scan&0x0f);
             // 输出BCD码(高4位)与扫描信号(低4位)
             delay1m(4);                     // 延迟4ms
             scan>>=1;                       // 产生下个扫描信号
         }                                   // 结束扫描4个数字j循环
     }                                       // 结束扫描30次i循环
}                                            // scanner函数结束
```

使用7447译码器的四位数7段LED数码管驱动程序（ch5-1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图5-27连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 请将程序中的延迟时间缩短为原来的1/4，结果会如何？同样地，请将程序中的延迟时间缩短为原来的1/10，结果会如何？

2. 在本实验里，数字显示是由最左边开始扫描循序显示；请修改程序，让数字显示由最右边开始扫描循序显示？其显示结果，看得出来吗？

3. 使用两个四位数的7段LED数码管模块，连接成8个数字的显示电路，再使用74138与7447作为译码之用，如图5-30所示，若要显示“12345678”程序应如何编写？

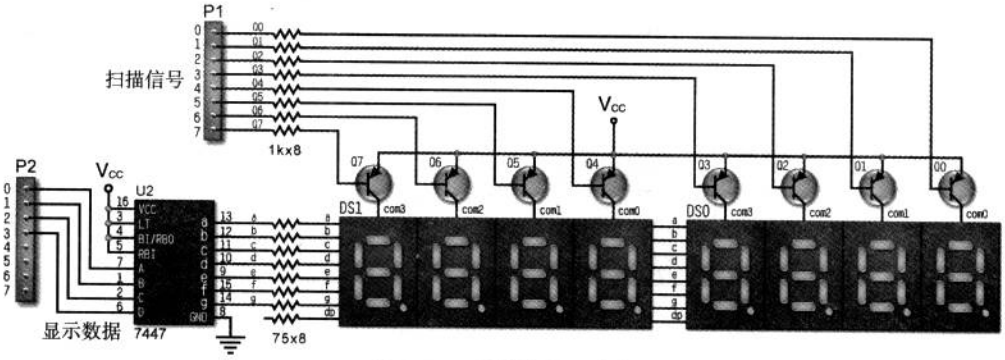


图5-30 八位数显示电路

5.5.2 直接驱动7段LED数码管实例演练

实验要点

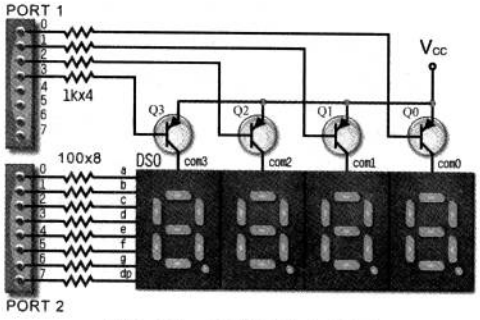
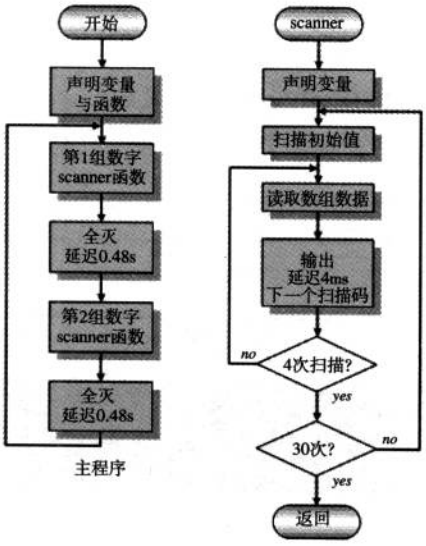


图5-31 直接驱动电路图

如图5-31所示，由P2将所要显示的7段LED数码管直接输出到4位数字的7段LED数码管模块，再由P1的低四位将扫描信号直接分送到7段LED数码管模块的4个共同端，使这个7段LED数码管模块闪烁“2008”3次，再闪烁“0315”3次，如此循环不停。

流程图与程序设计

本实验的电路比5.5.1节实验电路简单，而程序设计也比5.5.1节实验的程序设计简单！主要是因为扫描信号与显示数字信号分开了。当然，不使用外部译码器，就得在程序里建构一个BCD编码表，储存在数组里，经过转码后，就可输出到7段LED数码管模块。而扫描信号也可直接输出，不必再处理。而且，主程序与5.5.1节的主程序一样，只要在scanner函数里稍微修改即可。




```
/* ch5_2.c - 直接驱动4位7段LED数码管实验，P1.0~3 为扫描信号 P2接7段LED数码管 */
//==声明区=====================================
#include   <reg51.h>        // 定义8051寄存器的头文件，P2-17~19
char TAB[10]={ 0xc0，0xf9，0xa4，0xb0，0x99，        //数字0-4
               0x92，0x83，0xf8，0x80，0x98 };      //数字5-9
char disp[2][4]={ {2，0，0，8},          // 显示数据(第0组)
                  {0，3，1，5}};         // 显示数据(第1组)
void delay1m(int);                       // 声明延迟函数
void scanner(char);                      // 扫描函数
//==主程序=====================================
main()                                   // 主程序开始
{    char i;                             // 声明变量i
     while(1)                            // 无穷循环，程序一直运行
     {    for(i=0；i<2；i++)             // 显示第0，1列字组，for循环(字组i)开始
          {   scanner(i);                // 扫描第i列字组
              P1=0xff;                   // 全灭
              delay1m(480);              // 延迟480×1m=0.48s
          }                              // for循环(字组i)结束
     }                                   // while循环结束
}                                        // 主程序结束
//==子程序=====================================
/* 延迟函数，延迟约x×1ms */
```

```
void delay1m(int x)                    // 延迟函数开始
{   int i, j;                          // 声明整数变量i, j
    for (i=0; i<x; i++)                // 计数x次, 延迟x×1ms
        for (j=1; j<=120; j++);        // 计数120次, 延迟1ms
}                                      // 延迟函数结束
/* 扫描字组函数, 显示第x组数字 */
void scanner(char x)                   // 扫描字组函数开始
{   char i, j, BCD, scan;              // 声明变量
    for (i=0; i<30; i++)               // 扫描30次i循环
    {   scan=0xf7;                     // 扫描信号初值1111 0111
        for (j=0; j<4; j++)            // 扫描4个数字j循环
        {   P2=0xff;                   // 关闭7段显示器(防止闪动)
            P1=scan;                   // 输出扫描信号(低4位)
            BCD=disp[x][j];            // 读取第x组第j个数字的BCD码
            P2=TAB[BCD];               // 输出至7段LED数码管
            delay1m(4);                // 延迟4ms
            scan>>=1;                  // 产生下个扫描信号
        }                              // 结束扫描4个数字j循环
    }                                  // 结束扫描30次i循环
}                                      // scanner函数结束
```

直接驱动7段LED数码管实验（ch5-2.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图31连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，为何每组数字要连续扫描30次？试改变为60次或10次，各试一次，看看有何变化？

2. 在本实验里，数字显示是由最右边开始扫描循序显示；请修改程序，让数字显示由最左边开始扫描循序显示？其显示结果，看得出来吗？

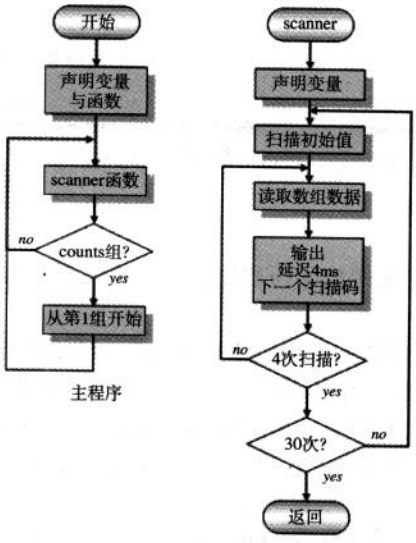
5.5.3 跑马灯实例演练

实验要点

电路图同图5-31，在此将以跑马灯的方式，将“123456”数字由右边依序走入四位数的7段LED数码管模块。即“----”→“---1”→“--12”→“-123”→“1234”→“2345”→“3456”→“456-”→“56--”→“6---”，如此循环不停。

流程图与程序设计

基本上，本实验的设计方式与5.5.2节一样，所不同的是显示的数字组的变化与组数的不同而已。主程序修改成可以显示多组数字的方式，scanner函数则完全一样。数字组从“----”变化到“6---”，总共10组字，而“-”的编码为“10111111”，即0xbf，在此将它放入TAB[10]，也就是驱动信号数组的最后一个位置。另外，在此使用#define预处理，将数字组数counts定义为10，以后如果要改变数字组数，只要在#define预处理修改即可。




```
    /* ch5_3.c - 4个7段LED数码管跑马灯实验，P1.0~3 为扫描信号 P2接7段LED数
码管 */
    //==声明区=====================================
    #include <reg51.h>   // 定义8051寄存器的头文件
    char TAB[11]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,                    // 数字0-4
                   0x92, 0x83, 0xf8, 0x80, 0x98, 0xbf };            // 数字5-9, -
```

```
#define counts 6                          // 声明字组数量
char disp[counts+7]={10, 10, 10, 10, 1, 2, 3, 4, 5, 6, 10, 10, 10};
//----123456---
void delay1m(int);                        // 声明延迟函数
void scanner(char);                       // 扫描函数
//==主程序=====================================
main()                                    // 主程序开始
{   char i;                               // 声明变量i
    whilev1)                              // 无穷循环，程序一直运行
      for(i=0; i<counts; i++)             // 显示count列字组，for循环(字组i)开始
        scanner(i);                       // 扫描第i列字组
}                                         // 主程序结束
//==子程序=====================================
/* 延迟函数，延迟约x*1ms */
void delay1m(int x)                       // 延迟函数开始
{   int i, j;                             // 声明整数变量i, j
    for (i=0; i<x; i++)                   // 计数x次，延迟x*1ms
      for vj=1; j<=120; j++);             // 计数120次，延迟1ms
}                                         // 延迟函数结束
/* 扫描字组函数，显示第x组数字 */
void scanner(char x)                      // 扫描字组函数开始
{   char i, j, BCD, scan;                 // 声明变量
    for (i=0; i<30; i++)                  // 扫描30次i循环
    {   scan=0xf7;                        // 扫描信号初值1111 0111
        for (j=0; j<4; j++)               // 扫描4个数字j循环
        {   P2=0xff;                      // 关闭7段显示器(防止闪动)
            P1=scan;                      // 输出扫描信号(低4位)
            BCD=disp[x+j];                // 读取第x组第j个数字的BCD码
            P2=TAB[BCD];                  // 输出至7段LED数码管
            delay1m(4);                   // 延迟4ms
            scan>>=1;                     // 产生下个扫描信号
        }                                 // 结束扫描4个数字j循环
    }                                     // 结束扫描30次i循环
}                                         // scanner函数结束
```

跑马灯实验程序（ch5-3.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图5-31连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 请修改本实验的程序，使跑马灯所显示的数字为“12345678”？

2. 请修改本实验的程序，使跑马灯走的方向为由左到右？

3. 请修改本实验的程序，让数字一个一个由右边“飞入”7段LED数码管模块？

4. 请修改本实验的程序，先让数字一个一个由右边“飞入”7段LED数码管模块，4个数字到齐后，再由右边一个一个“飞出”？

5.5.4 4×4键盘与7段LED数码管实例演练

实验要点

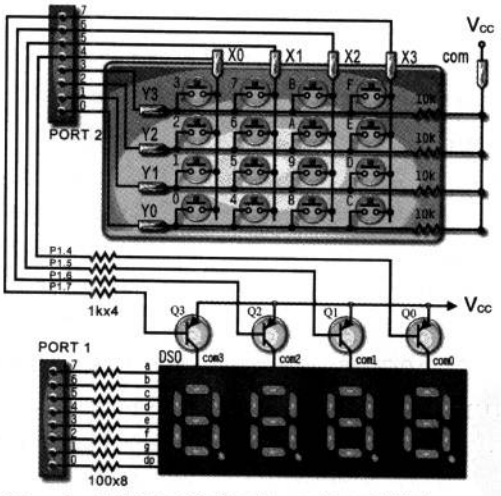


图5-32 4×4键盘与4位数7段LED数码管模块实验电路图

如图5-32所示，Port 2连接到四位数7段LED数码管模块的a、b、c…g。而Port 1的高四位提供扫描信号，P1.4连接到4×4键盘的X0及7段LED数码管模块的digit 0；P1.5连接到4×4键盘的X1及7段LED数码管模块的digit 1；P1.6连接到4×4键盘的X2及7段LED数码管模块的digit 2；P1.7连接到4×4键盘的X3及7段LED数码管模块的digit 3。Port 1的低四位连接到4×4键盘的Y3、Y2、Y1及Y0。我们所按下的键值将由左到右显示在7段LED数码管模块里。

流程图与程序设计

整个程序包括两个功能，第一个功能是显示数字，在此将所要显示的数据放在显示区数组里，刚开始时显示区数组里为空白的显示信号。每次扫描时，即显示一个位数，如图5-33所示。

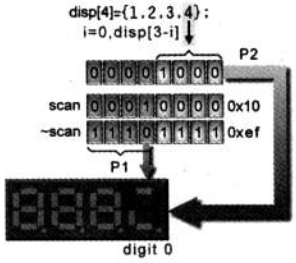


图5-33 输出显示区数据示意图

该程序的第二个功能是扫描键盘与判断，送出一个扫描信号后，即读入该行的键盘状态，然后逐列判断，若判断到按键被按下，就将其键值依序填入显示区数组里（由左而右），若数组显示区满了，则从最左边开始填入，如图5-34所示。

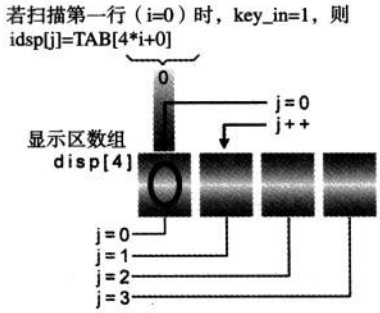
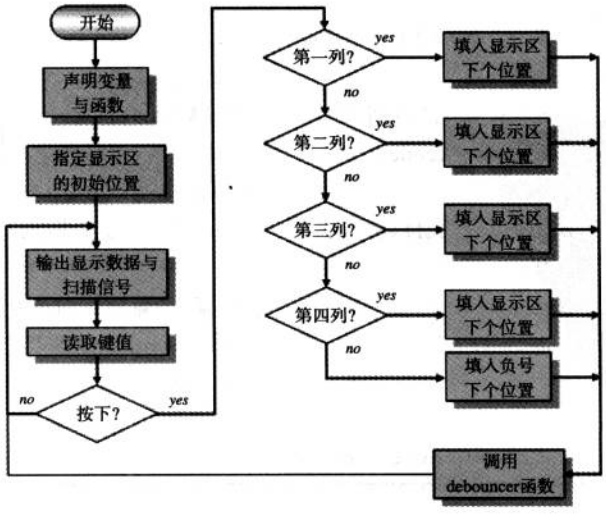


图5-34 填入显示区示意图

整个程序的流程图如下：



```
/* ch05_4.c - 4X4键盘与4个7段LED数码管实验，P1.4~7为共享扫描信号 */
/*      P1.0~3为键盘输入值，P2为7段LED数码管直接输出 */
//==声明区============================
#include <reg51.h>      // 定义8051寄存器的头文件
/* 声明7段LED数码管驱动信号数组(共阳) */
char TAB[18]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,        // 数字0-4
               0x92, 0x82, 0xf8, 0x80, 0x98,        // 数字5-9
               0xa0, 0x83, 0xa7, 0xa1, 0x84,        // 字母a-e(10-14)
               0x8e, 0xbf, 0x7f };   // 字母F(15)，负号(-)，小数点(.)
char disp[4]={ 0x7f, 0x7f, 0x7f, 0x7f };  // 声明4位7段LED数码管数组初值为小
                                          数点
char dig=0;                 // 声明7段LED数码管写入第几位(初值为最右边)
void debouncer(void);  // 声明防抖动函数
void scanner(void);       // 声明扫描函数
#define rowkey() (~P1)&0x0f              // 读入P1低4位(列按键值)宏
//==主程序============================
main()                                   // 主程序开始
{   while(1)                             // 无穷循环，程序一直运行
        scanner();                       // 扫描函数
}                                        // 主程序结束
//==子程序============================
/* 防抖动函数，延迟约40ms */
void debouncer(void)                     // 防抖动函数开始
{   int i;                               // 声明整数变量i
    for(i=0; i<4800; i++);               // 计数4800次，延迟约40ms
}                                        // 防抖动函数结束
/* 扫描4×4键盘及4个7段LED数码管函数 */
void scanner(void)                       // 扫描函数开始
{   char col, row;      // 声明变量(目前扫描第col行，第row列)
    char scan, keyin, kcode;             // 声明变量(扫描信号，列按键值，扫描码)
    scan=0xef;                           // 扫描信号初值=1110 1111
    for(col=0; col<4; col++)             // for循环，扫描第col行
    {   P2=0xff;                         // 关闭7段LED数码管
        P1=scan;                         //P1高4位输出扫描信号，低4位设为输入
        P2=disp[col];                    // 输出第col行数字
        keyin=rowkey();                  // 读入P1低4位(第col行的列按键值)
        if(keyin != 0)                   // 若有按键时
        {    for(row=0; row<4; row++)  // 扫描第row列是否按下
```

```
            {   if(keyin == (0x01<<row)) // 只有第row列按下
                { kcode=row+4*col;       // 计算出按键码为
                  disp[dig]=TAB[kcode]; // 扫描码写入7段LED数码管数组
                  dig=(dig<3)? dig+1:0; // 未满显示左1位，满时至最右位
                  break;                // 离开第row列扫描(for循环)
                }                       // if语句(第row列按下)结束
                if (row==4)             // 输入错误时
                { disp[dig]=TAB[16];    // 负号(-)写入7段LED数码管数组
                  dig=(dig<3)? dig+1:0; // 未满显示左1位，满时至最右位
                }                       // if语句(输入错误)结束
            }                           // for循环，扫描row列结束
            while(rowkey() !=0);        // 等按钮放开
            debouncer();                // 防抖动函数
        }                               // if语句(有按键时)结束
        scan=(scan<<1)|0x01;            // 产生左移1位扫描信号
    }                                   // for循环，扫描col行结束
}                                       // 扫描函数结束
```

4×4键盘与7段LED数码管模块实验（ch5-4.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图5-32连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 若将本实验的程序中，不要调用防抖动子程序，会有什么结果？

2. 在本实验里，4×4键盘可不可以连接PORT 0？请详细描述。7段LED数码管可不可以接其他输入/输出口？

3. 若本实验电路中的PORT 1接错，使P1.0~P1.3与P1.4~P1.7对调，则程序要如何修改，才能有同样的功能？

5.5.5 MM74C922实例演练

实验要点

如图5-35所示，MM74C922的数据ABCD连接到8051的P2.4~P2.7，而P2.0~P2.3连接7447，以输出数字数据。MM74C922的DA引脚连接到8051的P3.2。若按下4×4键盘上的任一个键，该键的数字将显示在7段LED数码管上。

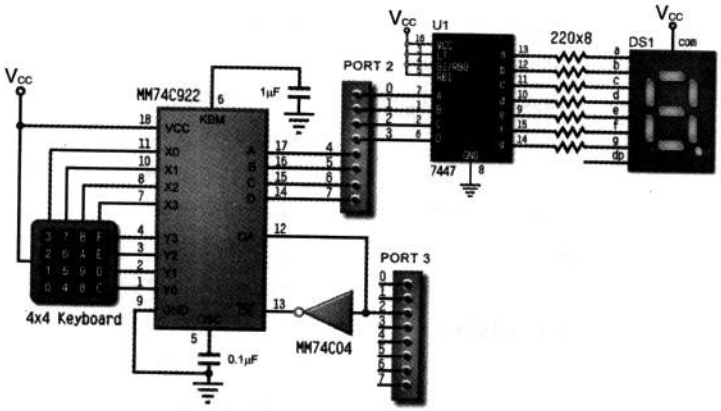
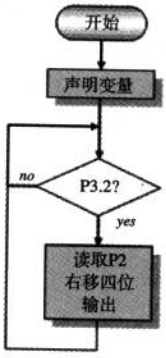


图5-35 应用74C922的电路图

流程图与程序设计

根据功能需求与电路结构可知，当键盘上有合法输入时，MM74C922将从DA引脚输出High，因此，P3.2引脚测得高电平时，随即读取2.4~P2.7的数据（即为键值）。程序只要持续判断DA引脚，作为是否读取键盘与显示数据的依据，整个流程图如下：




```
/* ch5_5.c - MM74C922实验，P2.4~7为键盘值，P2.0~3输出至7447 */
//==声明区=====================================
#include          <reg51.h>                  // 定义8051寄存器的头文件
sbit  IRQ=P3^2;                              // 声明P3.2 检测是否输入
//==主程序=====================================
```

```
main()                                    // 主程序开始
{    P2=0xff;              // 规划P2.4~7为输入口，并关闭显示器(P2.0~3)
     while(1)              // 无穷循环，程序一直运行
        if (IRQ==1)        // 判断是否按下按键
           P2=(P2>>4)|0xf0;
        /* 高4位(输入)移至低4位(输出)，再将高4位设为1(规划输入)*/
}                                         // 主程序结束
```

MM74C922实验（ch5-5.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图35连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，只是在等待按键，程序可否做其他事情？

2. 在本实验里，有没有“抖动”的困扰？

5.6 实时练习

本章内容以键盘扫描及7段LED数码管扫描为主，只要按部就班，即可成为扫描高手，更可成为7段LED数码管专家。在此请试着回答下列问题，以确认对于此部分的掌握程度。

选择题

（ ）1. 当我们要设计多位数7段LED数码管时，其扫描的时间间隔大约多少比较适当？
（A）0.45s （B）0.3s （C）0.15s （D）0.015s

（ ）2. 与多个单位数7段LED数码管比较，使用多位数的7段LED数码管模块具有何优点？
（A）数字显示比较好看 （B）成本比较低廉
（C）比较高级 （D）电路比较复杂

（ ）3. 若要连接4×4键盘与微处理器，至少需要多少位的输入/输出端口？
（A）16位 （B）12位 （C）9位 （D）8位

（ ）4. 对于多个按钮的输入电路而言，应如何连接比较简洁？
（A）采用数组式连接 （B）采用串行式连接
（C）采用并列式连接 （D）采用跳线式连接

（ ）5. 使用7447驱动7段LED数码管时，若要测试其所连接的7段LED数码管是否故障，应如何处理？
（A）将test引脚连接高电平 （B）将test引脚连接低电平
（C）将引脚连接高电平 （D）将引脚连接低电平

（ ）6. 使用74138译码时，应如何连接才能正常译码？
（A）G1、$\overline{G2A}$、$\overline{G2B}$引脚连接高电平
（B）G1、$\overline{G2A}$、$\overline{G2B}$引脚连接低电平
（C）G1引脚连接高电平，$\overline{G2A}$、$\overline{G2B}$引脚连接低电平
（D）G1引脚连接低电平，$\overline{G2A}$、$\overline{G2B}$引脚连接高电平

（ ）7. 74C922提供什么功能？
（A）7段LED数码管译码功能 （B）4×4键盘扫描
（C）4×5键盘扫描 （D）16位扫描信号产生器

（ ）8. TTL的输入引脚若悬空，将会如何？
（A）看作High （B）看作Low （C）高阻抗状态 （D）不允许

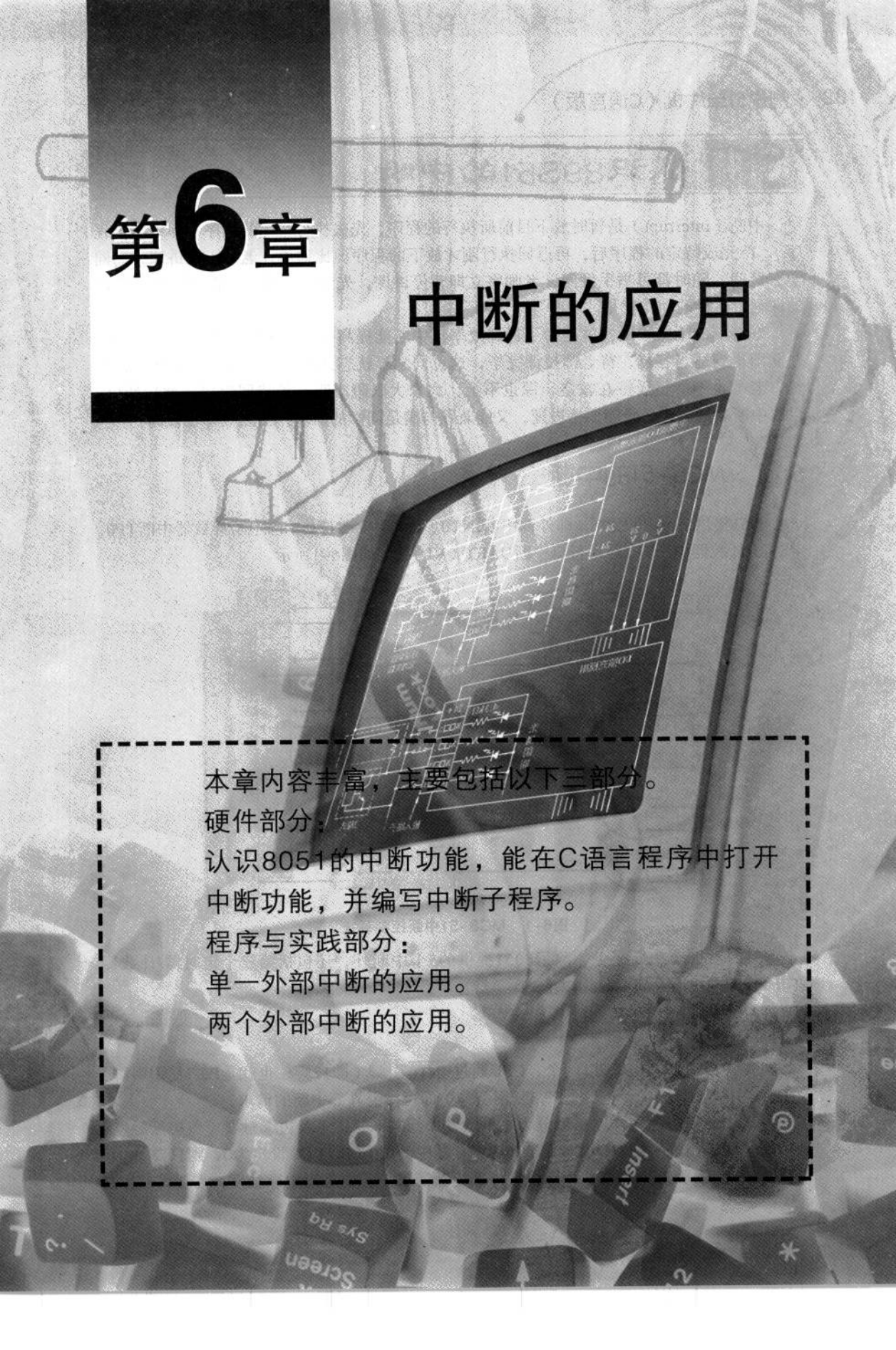
（ ）9. CMOS的输入引脚若悬空，将会如何？
（A）看作High （B）看作Low （C）高阻抗状态 （D）不允许

（ ）10. 7446与7447都是共阳极7段LED数码管译码驱动器，两者的差异为何？
（A）7446的驱动电流较大 （B）7447的驱动电流较大
（C）7446的负载电压较高 （D）7447的负载电压较高

问答题

1. 试说明什么是“高电平扫描”？什么是“低电平扫描”？
2. 在MM74C922与MM74C923都是键盘扫描IC，其不同在哪里？
3. 试说明7446、7447、7448及7449的异同？
4. 试说明74138及74139的不同？
5. 试画出如何用16个Tack Switch连接成4×4键盘？

第6章 中断的应用



本章内容丰富，主要包括以下三部分。

硬件部分：

认识8051的中断功能，能在C语言程序中打开中断功能，并编写中断子程序。

程序与实践部分：

单一外部中断的应用。

两个外部中断的应用。

6.1 认识89S51的中断

中断（interrupt）是暂时放下目前所执行的程序，先去执行特定的程序（即中断子程序），待完成特定的程序后，再返回执行刚才放下的程序。比如说，老师正在讲课，而同学有疑问，随时都可举手发问，老师将立即暂停讲课，先为同学解惑，再继续刚才暂停的课程，这样的动作就是“中断”。

好端端的干嘛中断？就是为了提高效率！中断能提高效率？试想若不立即提出问题并立即得到适当答复，待老师授课完毕，这位同学早就忘光了，同时也失去兴趣了！当然，老师也不能整天待在教室，课也不上，大家大眼瞪小眼，等待同学提问题！所以，以“中断”的方式，既能保持进度，又能兼顾与满足同学的需求，当然效率比较高！

6.1.1 MCS-51的中断

8051提供5个中断服务，即外部中断INT0、外部中断INT1、定时器/计数器中断TF0、定时器/计数器中断TF1与串行口中断UART（RI/TI），如图6-1所示。

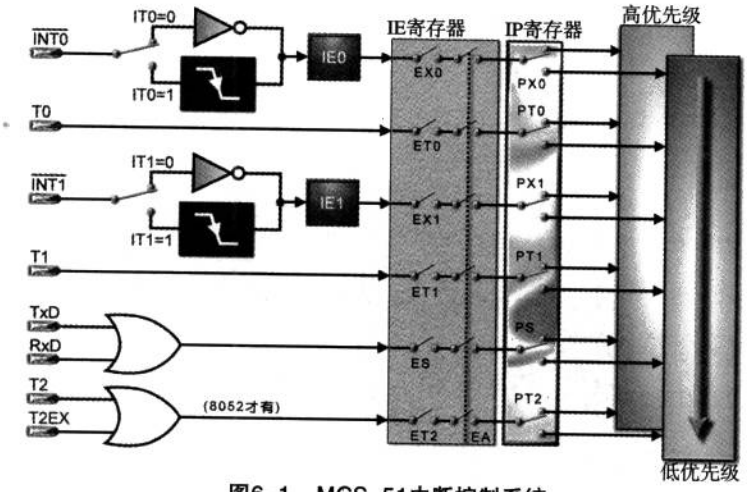


图6-1 MCS-51中断控制系统

8052提供6个中断服务，除了8051提供的5个中断外，还包括第三个定时器/计数器（Timer 2）的中断，大概分为以下3类。

外部中断

外部中断有INT0与INT1两个，CPU通过 $\overline{\text{INT0}}$ 引脚（即12脚，也就是P3.2复用引脚）及 $\overline{\text{INT1}}$ 引脚（即13脚，也就是P3.3复用引脚）即可接受外部中断的请求。

外部中断信号的采样方式可分为电平触发（低电平触发）及边缘触发（负边缘触发）两种。若要采用电平触发，需要将TCON寄存器中的IT0（或IT1）设定为0，则只要 $\overline{\text{INT0}}$ 引脚（ $\overline{\text{INT1}}$ 引脚）为低电平，即视为外部中断需求。若要采用边缘触发，需要将TCON

寄存器中的IT0（或IT1）设定为1，则只要 $\overline{\text{INT0}}$ 引脚（ $\overline{\text{INT1}}$ 引脚）的信号由高电平转为低电平，即视为外部中断需求。这些中断需求将反应在IE0（或IE1）里，若IE寄存器的EX0（或EX1）=1，且EA=1，CPU将进入该中断的服务。至于中断优先级寄存器（IP寄存器），只是安排多个中断发生时中断服务执行的顺序而已，若只有一个中断，将不会有所影响。

定时器/计数器中断

定时器/计数器中断有TF0与TF1两个（8052则还有TF2）。若是定时器，CPU将对内部的时钟脉冲计数，而提出内部中断；若是计数器，CPU将对外部的脉冲计数，而提出内部中断。至于外部脉冲的输入，则是通过T0引脚（即14脚，也就是P3.4复用引脚）及T1引脚（即15脚，也就是P3.5复用引脚）。关于定时器/计数器，将在第7章进行详细说明。

串行口中断

串行口中断（UART）有RI或TI两个，CPU通过RXD引脚（即10脚，也就是P3.0复用引脚）及TXD引脚（即11脚，也就是P3.1复用引脚）要求接收（RI）中断需求或传送（TI）中断需求。关于串行口，将在第8章进行详细说明。

当中断发生时，CPU将暂停当时所执行的程序，立即按中断的种类执行其中断向量。例如INT0中断时，程序将跳到03H地址（INT0的中断向量），而03H地址可能只有一个命令，也就是跳到该中断的服务程序的地址。当然，若用C语言编写程序，程序设计者难以察觉到地址的变化。当中断子程序执行完毕后，即可返回主程序，继续执行刚才中断时的下一个指令，如图6-2所示。

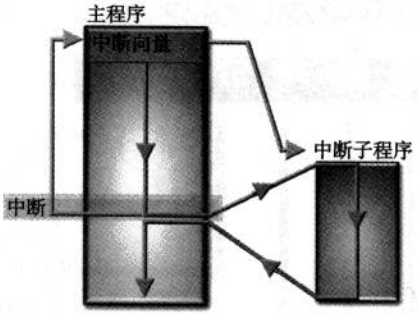


图6-2 中断流程

6.1.2 中断启用寄存器IE

如图6-1所示，我们可将中断启用寄存器（即IE寄存器）看作开闭中断功能的开关。实际上，IE寄存器是一个8位的可位寻址寄存器，其中各位如图6-3所示。

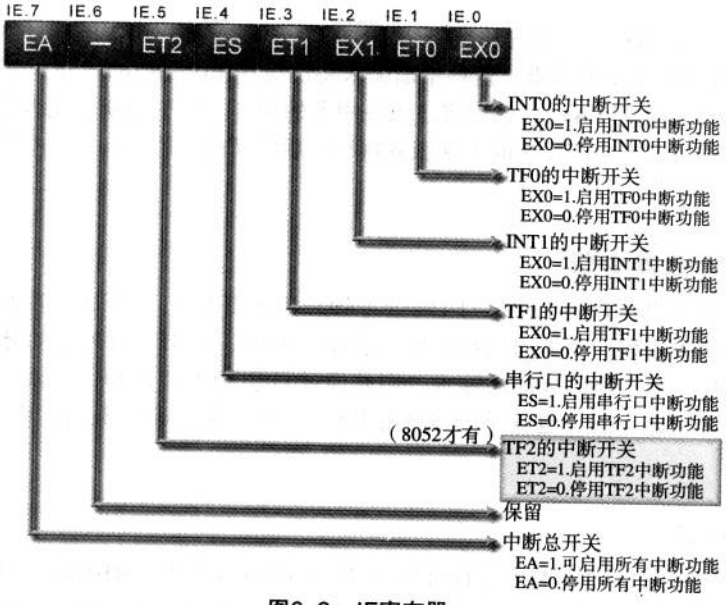


图6-3 IE寄存器

6.1.3 中断优先级寄存器IP

如图6-1所示，中断优先级寄存器（IP寄存器）是判断各中断优先级的开关。实际上，IP寄存器是一个8位的可位寻址寄存器，其中各位如图6-4所示。

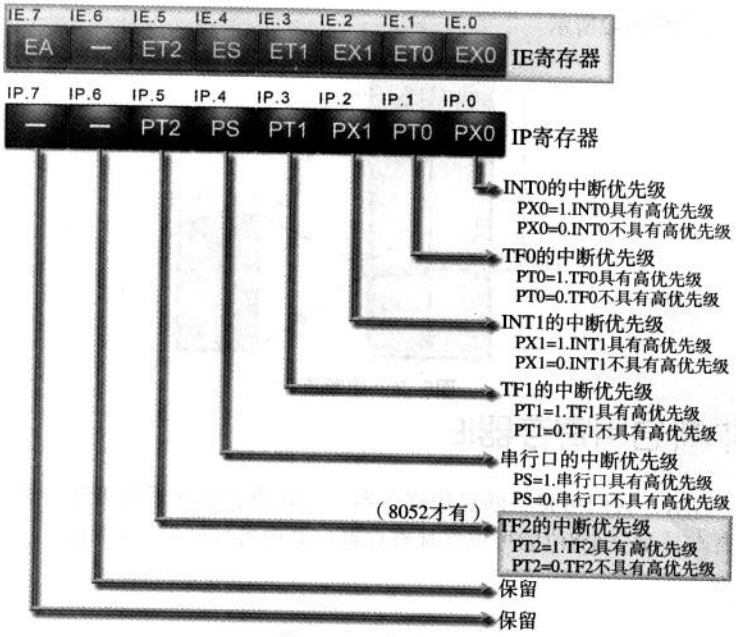


图6-4 IP寄存器

在此特将IE寄存器也放在上面，直接与IP寄存器比较，我们将可发现其中各位几乎是相对应的！所以记住了IE寄存器，也就记住了IP寄存器！

很明显，在图6-1中IP寄存器只是决定中断属“高优先级”族群，还是“低优先级”族群而已。原本各个中断已有先后之分，其顺序如图6-5所示。

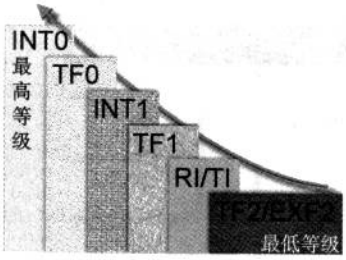


图6-5 预置优先级

如图6-5所示，若都没有在IP寄存器里设定优先级，则中断的优先级为“INT0>TF0>INT1>TF1>RI/TI>TF2/EXF2”；若将其中任一个中断设为高优先级，例如让TF1=1，则中断的优先级变为“TF1>INT0>TF0 >INT1>RI/TI>TF2/EXF2”；若让TF1=1，INT1=1，则中断的优先级变为“INT1>TF1>INT0>TF0>RI/TI>TF2/EXF2”，依次类推。图6-6所示为在不同优先级下程序执行的流程。

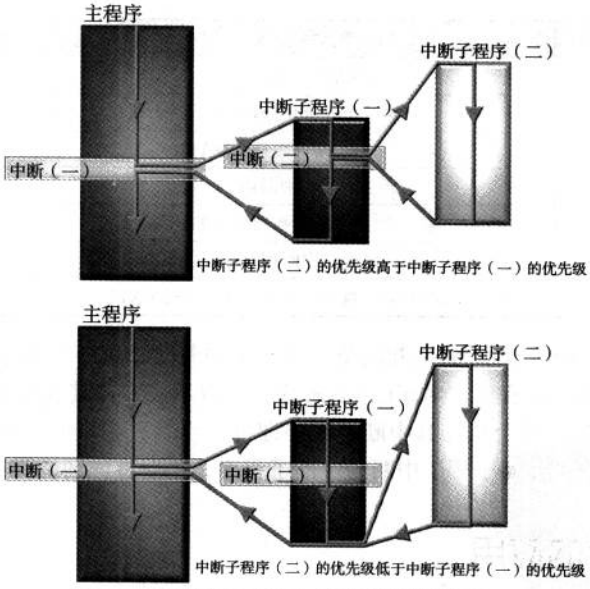


图6-6 不同优先级下，程序执行的流程

如图6-6所示，简单讲就是“优先级较低者站一边等”。

6.1.4 定时器/计数器控制寄存器TCON

在定时器/计数器控制寄存器TCON里，有部分设定与外部中断信号的采样方式有关。

如图6-7所示，TCON寄存器是一个8位的可位寻址寄存器。其中IT0与IT1分别为INT0与INT1的采样信号设置位，若要采用负边缘触发信号，则可将它设定为1；若要采用低电平动作信号，则可将它设定为0。IE0与IE1两个位是由CPU所操作的中断标志位，当中断发生时，将被设定为1；结束中断时，将恢复为0。

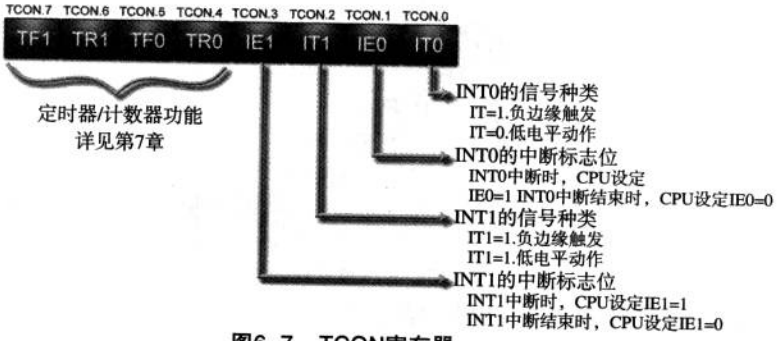


图6-7 TCON寄存器

6.1.5 中断向量

表6-1

中断编号	中断源名称	中断向量地址
–	系统复位（Reset）	0x00
0	第一个外部中断INT0	0x03
1	第一个定时器/计数器中断TF0	0x0B
2	第二个外部中断INT1	0x13
3	第二个定时器/计数器中断TF1	0x1B
4	串行口中断RI/TI	0x23
5	第三个定时器/计数器中断（8052）TF2/EXF2	0x2B

表6-1所示为8051/8052的中断向量。当发生中断时，程序将跳至其对应的中断向量地址，执行该位置上的程序。对于C语言程序，可以不必知道其真实地址，程序设计者只要知道发生中断时，将会执行其中断子程序即可。当然，在中断子程序的定义上，必须明确定义该中断子程序属于哪个中断的中断子程序，稍后将详细说明这个问题。

6.1.6 中断的应用

中断的应用包括中断向量的设置以及中断子程序的编写，说明如下。

中断设定

中断的设定包括开启中断开关（即IE寄存器的设定）、中断优先级的设定（即IP寄存器的设定）、中断信号的设定（即TCON寄存器的设定）等。我们可在程序里直接设定IE寄存器、IP寄存器及TCON寄存器，例如要开启“总开关”、“INT0开关”，可以使用以下命令。

```
IE=0x81;        // 启用INT 0中断
```

其中0x81就是10000001，相当于把IE寄存器中的EA与EX0设定为1。同理，若要开启“总开关”、“INT1开关”，可以使用以下命令。

```
IE=0x84;        // 启用INT 1中断
```

若要开启“总开关”、“INT0开关”及“INT1开关”，可以使用以下命令。

```
IE=0x85;        // 启用INT 0、INT1中断
```

对于中断优先级的设置，也使用类似的命令，只是操作的对象为IP寄存器，例如要提高INT1的优先级，可以使用以下命令。

```
IP=0x04;        // 设定INT 1中断具最高优先级
```

而外部中断信号的种类，可在TCON寄存器里设定，例如INT1中断要采用负边缘触发方式，可以使用以下命令。

```
TCON=0x08;  // 设定INT 1采用负边缘触发
```

● 中断子程序

中断子程序是一种特殊的子程序（函数），其第一行的格式如下。

```
void  中断子程序名称（void）  interrupt中断编号  using  寄存器组
```

其中的各项在第2章中已介绍过了，在此不赘述。例如要定义一个INT1（其中断编号为2）的中断子程序，其名称定义为“my_INT”，而在该中断子程序使用RB1寄存器组，则应做如下声明。

```
void  my_INT（void）  interrupt 2  using 1 // INT 1的中断子程序，使用RB1
```

接着，在一对大括号里，编写此中断子程序的内容。

6.2 中断子程序的仿真



在前几章中，我们已在Keil C的整合环境里进行过多次的仿真与调试，但尚未做过“中断”的仿真！当程序编辑完成并顺利通过编译后（可利用ch6-4-1.c范例程序操作），即可按下列的步骤进行中断仿真。

Step 1 按钮 打开调试工具栏，屏幕上出现确认对话框，如图6-8所示。



图6-8 确认对话框

Step 2 单击 确定 按钮关闭对话框，即可进入调试/仿真状态。再运行Peripherals菜单下的Interrupt命令，即可开启中断系统对话框（Interrupt System），如图6-9所示。

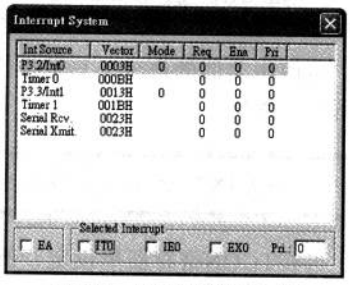


图6-9 中断系统对话框

Step 3 再次运行Peripherals菜单下的I/O-Ports命令，在打开的对话框中指定所要观察/追踪的输出入口（Port0 ~Port 3），即可开启该输出入口的对话框。

Step 4 单击 按钮，程序将进行仿真，主程序执行的结果将在输出入口的对话框中显示，同时在**Interrupt System**对话框下方，也将按程序的设定出现相对的选项。例如程序中有“IE=0x81”，则EA及EX0选项将自动选择；若程序中有“TCON=0x01”，则IT0选项将自动选取。若要让程序进入中断状态，则单击IE0选项，该选项将闪一下，程序即进入中断状态，我们就可从输出入口对话框中看到其操作。完成中断程序后，将自动恢复主程序的执行。当然，若要再次执行中断，还是一样指向IE0选项，单击鼠标左键即可。

6.3 实例演练

在本单元中提供了4个范例，以展示8051的外部中断功能。

6.3.1 外部中断INT0实例演练

实验要点

如图6-10所示，P2连接8个LED、第12脚（即INT0 $\overline{\text{INT0}}$ 引脚）连接一个10kΩ的上拉

电阻，让该引脚保持为High，另外再连接一个按钮开关（INT0）。当主程序正常执行时，P2所连接的8个LED将闪烁。若按INT0按钮开关，则进入中断状态，P2所连接的8个LED将变成单灯左移，而左移3圈（从最左边到最右边为1圈）后，恢复中断前的状态，程序将继续执行8灯闪烁的功能。

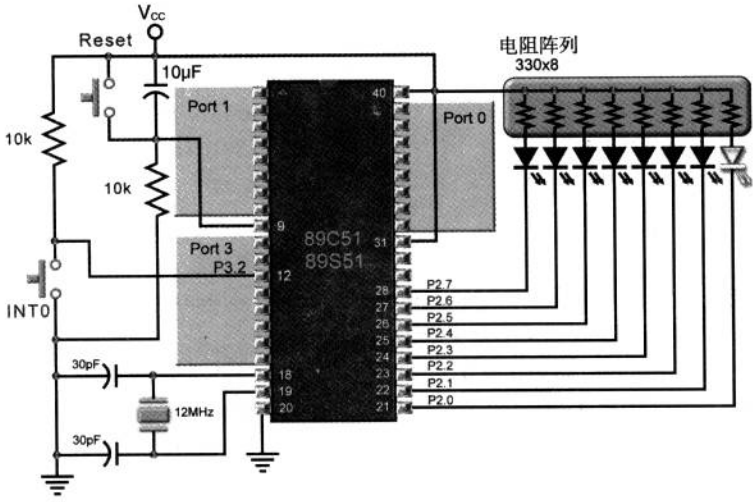
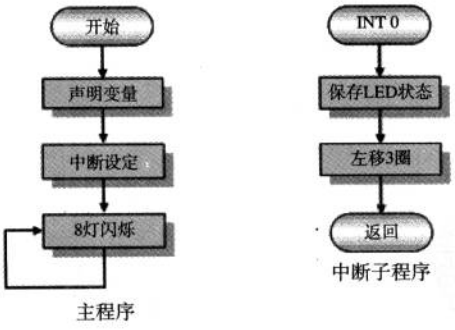


图6-10 INT0实验电路图

流程图与程序设计

根据功能需求与电路结构可知，首先声明delay函数，然后依次定义主程序，中断子程序与delay函数。在主程序里，先设定中断，然后进行8灯亮、延迟、8灯灭、延迟的持续动作。在中断子程序里，则以巢状循环的方式，内循环进行单灯左移8次，即可将亮灯由最右边移至最左边。外循环进行3次，也就是让单灯左移由最右边移至最左边，跑3圈后即可返回主程序。




```
/* ch6_1.c - INT0（Port3.2)中断实验 */
//==声明区=====================================
#include<reg51.h>                    // 定义8051寄存器的头文件
#define LED    P2                    // 定义LED接至Port 2
void delay(int);                     // 声明延迟函数
```

```
void left(int x);                          // 声明单灯左移函数
//==主程序==================================
main()                                     // 主程序开始
{    IE=0x81;                              // 准许INT 0中断
     TCON=0x01;                            // INT 0设为负边缘触发
     LED=0x00;                             // 初值=0000 0000，灯全亮
     while(1)                              // 无穷循环，程序一直运行
     { delay(250);                         // 延迟250×1m=0.25s
       LED=~LED;                           // LED反相
     }                                     // while循环结束
}                                          // 主程序结束
//==子程序==================================
/* INT 0的中断子程序 - 单灯左移3圈 */
void my_int0(void) interrupt 0             // INT0 中断子程序开始
{    unsigned saveLED=LED;                 // 储存中断前LED状态
     left(3);                              // 单灯左移3圈
     LED=saveLED;                          // 写回中断前ED状态
}                                          // 结束INT0 中断子程序
/* 延迟函数，延迟约x×1ms */
void delay(int x)                          // 延迟函数开始
{    int i, j;                             // 声明整数变量i, j
     for (i=0; i<x; i++)                   // 计数x次，延迟x×1ms
          for (j=1; j<120; j++);           // 计数120次，延迟1ms
}                                          // 延迟函数结束
/* 单灯左移函数，执行x圈 */
void left(int x)                           // 单灯左移函数开始
{    int i, j;                             // 声明变量i, j
     for(i=0; i<x; i++)                    // i循环，执行x圈
     { LED=0xfe;                           // 初始状态=1111 1110，最右灯亮
       for(j=0; j<7; j++)                  // j循环，左移7次
       {   delayv250 ) ;                   // 延迟250×1m=0.25s
           LED=vLED<<1)|0x01;              // 左移1位后，LSB设为1
       }                                   // j循环结束
       delay(250);                         // 延迟250×1m=0.25s
     }                                     // i循环结束
}                                          // 单灯左移函数结束
```

INT0中断实验（ch6-1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图6-10连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，中断子程序是以巢状循环方式，达到巢状单灯左移三圈的目的；若不要以巢状循环方式，请试着以其他方式达到同样的效果？

2. 在本实验里，若希望中断时这8个LED变成是霹雳灯，来回各三圈，才返回主程序，程序应如何更改？

6.3.2 外部中断INT1实例演练

实验要点

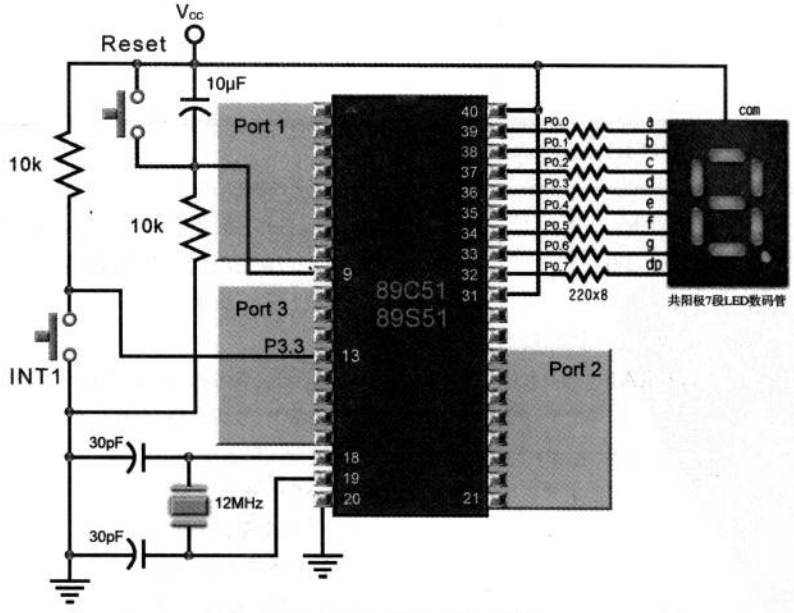


图6-11 INT1实验电路图

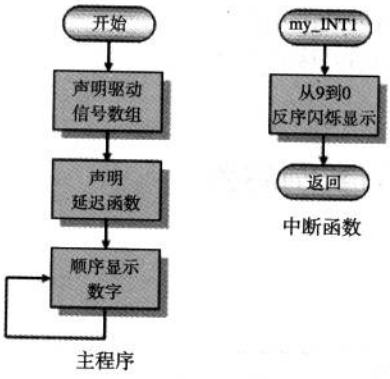
如图6-11所示，由P2直接驱动共阳极7段LED数码管；另外，第13脚（即 $\overline{\text{INT1}}$ 引脚）连接一个10kΩ的上拉电阻，让该引脚保持为High，再连接一个按钮开关（INT1）。当主程序正常执行时，7段LED数码管将从0开始正数到9（循环），每0.5s增加1。若按INT1按钮开关，则进入中断状态，则7段LED数码管将从9开始闪烁倒数到0（一圈后结束中断），每0.5s减少1。

● 流程图与程序设计

根据功能需求与电路结构可知，首先声明delay函数及驱动共阳极7段LED数码管的信号，我们可以把这些驱动信号保存为如下所示的字符数组。

```
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,
               0x92, 0x83, 0xf8, 0x80, 0x98 };
```

在主程序里，依次每隔0.5s输出字符数组中的编码，而在中断子程序里，反序每隔0.5s输出字符数组中的编码即可。




```
/* ch6_2.c - INT1(P3.3)中断实验 */
//==声明区=====================================
#include<reg51.h>                          // 定义8051寄存器的头文件，P2-17~19
#define SEG    P0                          // 定义7段LED数码管接至Port 0
void delay(int);                           // 声明延迟函数
/* 声明7段LED数码管驱动信号数组(共阳) */
unsigned char TAB[]={0xc0, 0xf9, 0xa4, 0xb0, 0x99, //数字0-4
                     0x92, 0x83, 0xf8, 0x80, 0x98 }; //数字5-9
//==主程序=====================================
main()                                     // 主程序开始
{    int i;                                // 声明i变量(计数值)
     IE=0x84;                              // 准许INT 1中断(6-4页)
     TCON=0x04;                            // INT 1设为负边缘触发(6-7页)
```

```
    while(1)                                    // 无穷循环，程序一直运行
    {   for(i=0; i<10; i++)                     // 显示0~9(上数)
        {   SEG=TAB[i];                         // 显示数字至7段LED数码管
            delay(500 ) ;                       // 延迟500×1m=0.5s
        }
    }                                           // for循环结束
}                                               // 主程序结束
//==子程序=====================================
/* INT 1的中断子程序 - 数字闪烁倒数9~0 */
void my_int1(void) interrupt 2                  // INT1 中断子程序开始
{   int i;                                      // 声明i变量(计数值 )
    for (i=9; i>=0; i--)                        // for循环显示9~0(下数)
    {   SEG=TAB[i];                             // 显示数字至7段LED数码管
        delay(250);                             // 延迟500×1m=0.5s
        SEG=0xff;                               // 关闭7段LED数码管
        delay(250);                             // 延迟500×1m=0.5s
    }                                           // for循环结束
}                                               // 结束中断子程序
/* 延迟函数，延迟约x×1ms */
void delay(int x)                               // 延迟函数开始
{   int i, j;                                   // 声明整数变量i, j
    for (i=0; i<x; i++)                         // 计数x次，延迟x×1ms
        for (j=1; j<120; j++);                  // 计数120次，延迟1ms
}                                               // 延迟函数结束
```

INT1中断实验（ch6-2.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。在此要特别注意，IE与TCON的设定不能错，因为即使它们设定错误照样能通过编译与链接！但功能却不一定会正常。另外，在中断子程序的开头“void my_int1（void）interrupt 2 using 1”，一定要符合IE与TCON的设定，例如“IE=0x84”代表INT 1中断，其中断编号为2，所以要定义为“**interrupt 2**”，千万不要搞错！

2. 在Keil C里进行软件调试/仿真，刚开始时，**Interrupt System**对话框下方显示“EA、IT0、IE0、EX0、Pri”5个项目，其中EA代表启用中断的总开关、IT0代表INT 0的触发信号种类、IE0为触动中断的信号、EX0代表启用INT 0的开关、Pri则为该中断的优先级。若要进行INT 1的中断，可指向**Interrupt System**对话框里的**P3.3/INT1**项，按鼠标左键；则对话框下方，IT 0项变成IT 1项、IE 0项变成IE 1项、EX 0项变成EX 1项，即可进行INT 1的中断，如图6-12所示。此后，程序进行时，只要指向IE 1项，按鼠标左键，即可进入中断子程序。

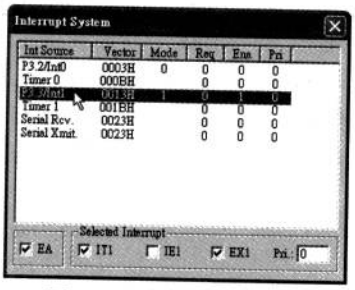


图6-12 切换为Int1中断

3. 在本实验里，若要从仿真中看出其功能是否正确并不容易！若将延迟子程序里的数字增加20倍或更多，让输出变化的速度慢一点，则可看清楚其输出变化的顺序，再与驱动信号比较，即可验证是否符合功能。

4. 若软件调试/仿真功能正常，再按图6-11连接线路，使用在线仿真器（记得将延迟时间恢复正常），加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

5. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

6. 编写实验报告。

● 思考一下

1. 在本实验的仿真中，若要在输出口里看到0、1、2…9的变化，而非0~9的7段LED数码管驱动信号，应如何修改程序？

2. 若在本实验的电路里，将原本的共阳极7段LED数码管，改采用共阴极7段LED数码管，则程序应如何更改？

6.3.3 两个外部中断实例演练

● 实验要点

如图6-13所示，P2连接8个LED，第12脚（即 $\overline{\text{INT0}}$ 引脚）连接一个10kΩ的上拉电阻，让该引脚保持为High；再连接一个按钮开关（INT0）、第13脚（即 $\overline{\text{INT1}}$ 引脚）连接一个10kΩ的上拉电阻，让该引脚保持为High，另外再连接一个按钮开关（INT1）。当主程序正常执行时，P2所连接的8个LED将闪烁。若按INT0按钮开关，则进入INT0中断状态，P2所连接的8个LED将变成单灯左移，而左移3圈（从最左边到最右边为1圈）后，恢复中断前的状态，程序将继续执行8灯闪烁的功能。若按INT1按钮开关，则进入INT1中断状态，P2所连接的8个LED将变成单灯右移，而右移3圈（从最左边到最右边为1圈）后，恢复中断前的状态，程序将继续执行8灯闪烁的功能。另外，在此要求单灯左移（INT0）中断的优先级较单灯右移（INT1）中断的优先级高。

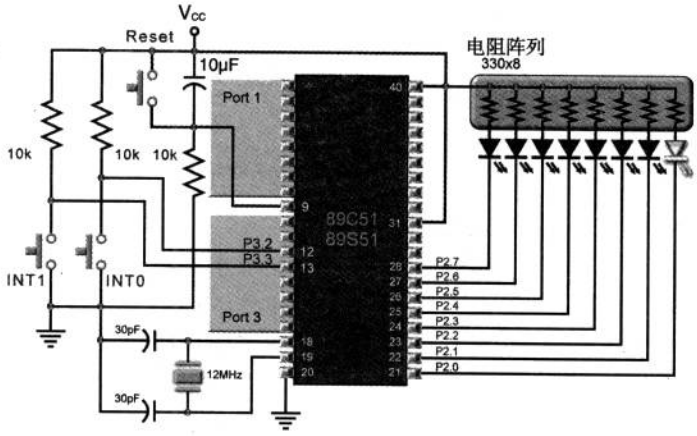
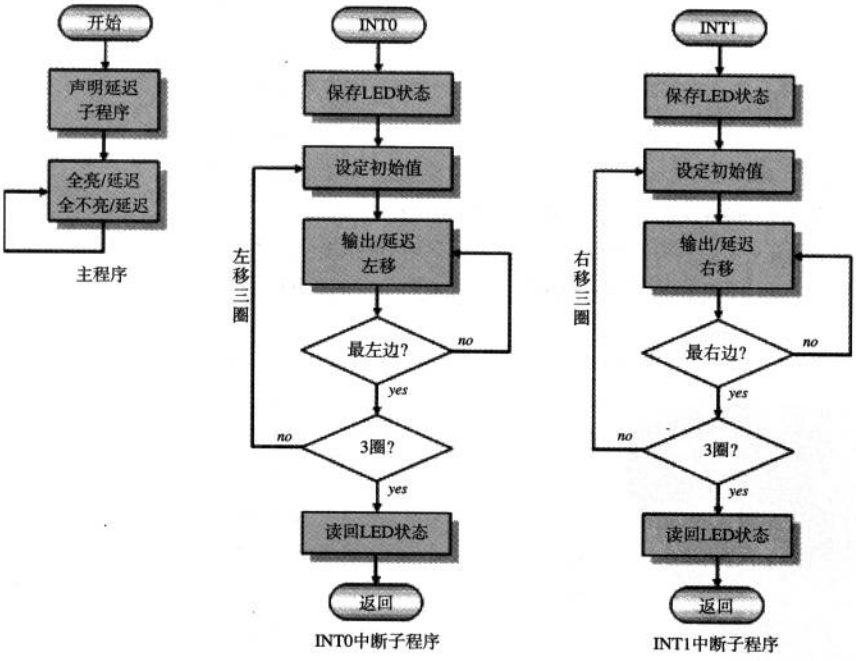


图6-13 两个外部中断实验电路图

流程图与程序设计

根据功能需求与电路结构可知，首先声明delay函数，然后在主程序里，依次每隔0.25s改变输出的亮/不亮状况，而且是持续不断地进行这样的操作。在INT0中断子程序里，进行每隔0.5sLED左移输出，由右而左执行3圈后，返回主程序；在INT1中断子程序里，进行每隔0.5sLED右移输出，由左而右执行3圈后，返回主程序。




```
/* ch6_3.c - 两个外部中断实验，INT0(P3.2)，INT1(P3.3) */
//==声明区=====================================
```

```
#include<reg51.h>                          // 定义8051寄存器的头文件
#define LED    P2                          // 定义LED接至Port 2
void delay(int);                           // 声明延迟函数
void left(int x);                          // 声明单灯左移函数
void right(int x);                         // 单灯右移函数开始
//==主程序==================================
main()                                     // 主程序开始
{    IE=0x85;                              // 准许INT 0，INT 1中断
     IP=0x01;                              // 设定INT 0具有最高优先级
     TCON=0x05;                            // INT 0，INT 1设为负边缘触发
     LED=0x00;                             // 初值=0000 0000，灯全亮
     while(1)                              // 无穷循环，程序一直运行
     {    delay(250);                      // 延迟250×1m=0.25s
          LED=~LED;                        // LED反相
     }                                     // while循环结束
}                                          // 主程序结束
//==子程序==================================
/* INT 0的中断子程序 - 单灯左移3圈 */
void my_int0(void) interrupt 0             // INT0 中断子程序开始
{    unsigned saveLED=LED;                 // 储存中断前LED状态
     left(3);                              // 单灯左移3圈
     LED=saveLED;                          // 写回中断前ED状态
}                                          // 结束INT0 中断子程序
/* INT 1的中断子程序 - 单灯右移3圈 */
void my_int1(void) interrupt 2             // INT1 中断子程序开始
{    unsigned saveLED=LED;                 // 储存中断前LED状态
     right(3);                             // 单灯左移3圈
     LED=saveLED;                          // 写回中断前ED状态
}                                          // 结束INT0 中断子程序
/* 延迟函数，延迟约x×1ms */
void delay(int x)                          // 延迟函数开始
{    int i，j;                             // 声明整数变量i，j
     for (i=0；i<x；i++)                   // 计数x次，延迟x×1ms
          for (j=1；j<120；j++);           // 计数120次，延迟1ms
}                                          // 延迟函数结束
/* 单灯左移函数，执行x圈 */
void left(int x)                           // 单灯左移函数开始
{    int i，j;                             // 声明变量i，j
```

```
    for(i=0; i<x; i++)                    //i循环，执行x圈
    { LED=0xfe;                           //初始状态=1111 1110，最右灯亮
      for(j=0; j<7; j++)                  //j循环，左移7次
      { delay(250);                       //延迟250×1m=0.25s
        LED=(LED<<1)|0x01;                //左移1位后，LSB设为1
      }                                   //j循环结束
      delay(250);                         //延迟250×1m=0.25s
    }                                     //i循环结束
}                                         //单灯左移函数结束
/* 单灯右移函数，执行x圈 */
void right(int x) //单灯右移函数开始
{   int i, j;                             //声明变量i, j
    for(i=0; i<x; i++)                    //i循环，执行x圈
    { LED=0X7f;                           //初始状态=0111 1111
      for(j=0; j<7; j++)                  //j循环，右移7次
      { delay(250);                       //延迟250×10m=0.25s
        LED=(LED>>1)|0x80;                //左移1位后，MSB设为1
      }                                   //j循环结束
      delay(250);                         //延迟250×1m=0.25s
    }                                     //i循环结束
}                                         //单灯左移函数结束
```

两个外部中断实验（ch6-3.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。其中IE、IP与TCON的设定千万不要弄错。另外，在每个中断子程序的开头，也一定要配合IE与TCON的设定。

2. 在Keil C里进行软件调试/仿真时，必须开启**Interrupt System**对话框，然后在对话框中点选**P3.3/INT1**项，下方显示“EA、IT1、IE1、EX1、Pri”5个项目，即可仿真INT1的动作。同样在对话框中选择**P3.2/INT0**项，下方显示“EA、IT0、IE0、EX0、Pri”5个项目，即可仿真INT0的动作。

3. 若软件调试/仿真功能正常，再按图6-13连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，“IP=0x01”表示INT0的中断优先级较高，则进行调试/仿真时，先INT0中断，LED由右而左3圈；若还没结束INT0之前，启动INT1中断，将会有什么变化？同样地，先INT1中断，LED由左而右3圈；若还没结束INT1之前，启动INT0中断，将会有什么变化？请记录在实习报告里。

2. 若希望INT1中断的优先级高于程序INT0中断的优先级，应如何修改？修改完成后，再进行一次第1题的实验，并记录在实习报告里。

6.3.4 键盘中断实例演练

实验要点

如图6-14所示，MM74C922（详见5.1.2节）的数据ABCD连接到8051的P2.4到P2.7，而P2.0到P2.3连接7447，以输出数字数据。MM74C922的DA引脚经过非门（MM74C04）连接到8051的P3.2，也就是INT0引脚。另外，8051的PORT 0连接8个LED。当主程序正常执行时，8个LED闪烁（每0.1s切换一次），若按下4×4键盘上的任一个键，则该键的数字将显示在7段LED数码管上，而PORT 0所连接的LED，仍保持正常的闪烁。

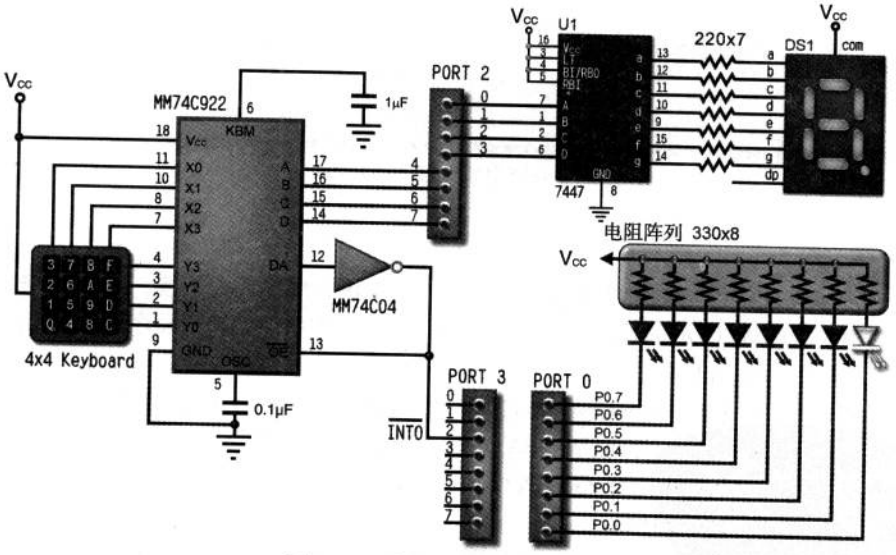
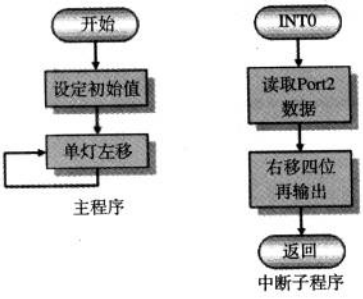


图6-14 键盘中断实验电路图

流程图与程序设计

根据功能需求与电路结构可知，首先声明INT0中断，然后在主程序里即进行无穷尽的单灯左移。在INT0中断子程序里，首先读取Port 2的数据，然后将该数据右移4次，让高四位移到低四位来，再将该数据输出到Port 2即可。




```
/* ch6_4.c - 4x4键盘实验，外部INT0(P3.2) */
//==声明区=====================================
#include<reg51.h>                  // 定义8051寄存器的头文件
#define LED    P0                  // 定义LED接至Port 0
#define KEYBCD  P2   // 定义接键盘扫描(低4位)及BCD输出(高4位)
#define RR(x)  (x==0x7f)? 0xfe:(x<<1)|0x01     //x向左旋转1bit
void delay(int);                   // 声明延迟函数
//==主程序=====================================
main()                             // 主程序开始
{    IE=0x81;                      // 准许INT 0中断
     TCON=0x01;                    // INT 0设为负边缘触发
     LED=0xfe;                     // 初值=1111 1110，最右灯亮
     while(1)                      // 无穷循环，程序一直运行
     {   delay(250);               // 延迟250 × 1m=0.25s
         LED=RR(LED);              // LED向左旋转1bit
     }                             // while循环结束
}                                  // 主程序结束
//==子程序=====================================
/* 延迟函数，延迟约x × 1ms */
void delay(int x)                  // 延迟函数开始
{    int i, j;                     // 声明整数变量i, j
     for (i=0; i<x; i++)           // 计数x次，延迟x × 1ms
          for (j=1; j<120; j++);   // 计数120次，延迟1ms
}                                  // 延迟函数结束
/* INT 0的中断子程序 - 单灯左移3圈 */
void my_int0(void) interrupt 0     // INT0 中断子程序开始
{    unsigned saveLED=LED;         // 储存中断前LED状态
     KEYBCD=0xff;                  // 规划输入
     KEYBCD>>=4;                   // 高四位移至低四位
```

```
    LED=saveLED;                    // 写回中断前ED状态
}                                   // 结束INT0 中断子程序
```

4×4键盘实验（ch6-4.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 请按图6-11连接线路，再使用在线仿真器，加载该程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

3. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

4. 编写实验报告。

思考一下

1. 在本实验里，有没有“抖动”的困扰？
2. 在本实验里，若按住按键不放会怎样？

6.4 实时练习



在本章中探讨了8051的中断，包括中断向量、与中断相关的寄存器、中断程序的编写等，这些可以说是微机系统中不可或缺的部分。在此请试着回答下列问题，以确认对于此部分的掌握程度。



（　）1. 中断功能具有什么好处？
（A）让程序更复杂　（B）让程序执行速度更快
（C）让程序更有效率　（D）以上皆非

（　）2. 8051提供几个外部中断？几个定时器/计数器中断？
（A）2，2　（B）3，6　（C）2，3　（D）3，7

（　）3. 8051的IP寄存器的功能是什么？
（A）设定中断优先级　（B）启用中断功能
（C）设定中断触发信号　（D）定义CPU的网址

（　）4. 若要让 $\overline{\text{INT0}}$ 采用低电平触发，则应如何设定？
（A）EX0=0　（B）EX0=1　（C）IT0=0　（D）IT0=1

（　）5. 在Keil μVision 3里进行调试/仿真时，可在那里操作，才能触动程序中断？

（A）在Interrrupt System对话框 （B）在Cotrol Box对话框里
（C）直接单击 按钮即可 （D）直接按 0 键

（ ）6. 8051所提供的中断功能里，下列哪个优先级较高？
（A）T1 （B）RI/TI （C）T0 （D）INT0

（ ）7. 在TCON寄存器里，IE1的功能是什么？
（A）触发INT1中断 （B）指示INT1中断的标志位
（C）提高INT1优先级 （D）取消INT1中断

（ ）8. 在Keil C里，中断子程序与函数有何不同？
（A）中断子程序不必声明 （B）函数不必声明
（C）中断子程序必须有自变量传入 （D）中断子程序一定会有自变量传出

（ ）9. 若要同时启用INT0及INT1中断功能，则应如何设定？
（A）TCON=0x81 （B）IE=0x85 （C）IP=0x83 （D）IE=0x03

（ ）10. 若要提高INT1的优先级，则可如何设定？
（A）IP=0x01 （B）IE=0x01 （C）IP=0x04 （D）IE=0x04

问答题

1. 试说明IE寄存器及IP寄存器中各位的功能。
2. 试说明MCS-51提供哪些中断，中断向量是什么？
3. 试说明如何设定IE寄存器。
4. 具有中断功能的程序，必须包含那些声明或设定？
5. 如何设定外部中断信号的种类？

（A）将Interrupt子函数的地址赋给T0

（B）将Interrupt子函数的地址赋给T1

（C）将Interrupt子函数的地址赋给T0

（D）将Interrupt子函数的地址赋给T1

3. 判断题

（1）51单片机的中断源有5个，分别是2个外部中断、2个定时/计数器中断和1个串行口中断。

（2）51单片机的中断优先级只有两级，即高级中断和低级中断。

（3）51单片机的中断嵌套最多为两级。

（4）51单片机的中断入口地址是固定的，不可以改变。

（5）51单片机的中断响应时间是固定的，不可以改变。

4. 简答题

（1）简述51单片机的中断响应过程。

（2）简述51单片机的中断返回指令RETI与普通返回指令RET有何区别？

（3）简述51单片机的中断嵌套规则。

5. 编程题

（1）设51单片机的晶振频率为12MHz，要求定时器T0工作在方式1下产生1ms的定时，试计算定时器T0的初值，并编写程序。

（2）设51单片机的晶振频率为12MHz，要求定时器T1工作在方式2下产生200μs的定时，试计算定时器T1的初值，并编写程序。

（3）设51单片机的晶振频率为12MHz，要求定时器T0工作在方式3下，TL0产生1ms的定时，TH0产生500μs的定时，试计算定时器T0的初值，并编写程序。

（4）设51单片机的晶振频率为12MHz，要求定时器T0工作在方式1下，在P1.0引脚上输出周期为2ms的方波，试计算定时器T0的初值，并编写程序。

（5）设51单片机的晶振频率为12MHz，要求定时器T0工作在方式2下，在P1.0引脚上输出周期为500μs的方波，试计算定时器T0的初值，并编写程序。

（6）设51单片机的晶振频率为12MHz，要求定时器T0工作在方式3下，TL0在P1.0引脚上输出周期为1ms的方波，TH0在P1.1引脚上输出周期为500μs的方波，试计算定时器T0的初值，并编写程序。

（7）设51单片机的晶振频率为12MHz，要求定时器T0工作在方式1下，在P1.0引脚上输出周期为2ms的方波，试计算定时器T0的初值，并编写程序。

（8）设51单片机的晶振频率为12MHz，要求定时器T0工作在方式2下，在P1.0引脚上输出周期为500μs的方波，试计算定时器T0的初值，并编写程序。

（9）设51单片机的晶振频率为12MHz，要求定时器T0工作在方式3下，TL0在P1.0引脚上输出周期为1ms的方波，TH0在P1.1引脚上输出周期为500μs的方波，试计算定时器T0的初值，并编写程序。

（10）设51单片机的晶振频率为12MHz，要求定时器T0工作在方式1下，在P1.0引脚上输出周期为2ms的方波，试计算定时器T0的初值，并编写程序。

（A）TCON=0x01　（B）IE=0x01　（C）IP=0x88　（D）IE=0x82

（A）IP=0x01　（B）IE=0x01　（C）IP=0x08　（D）IE=0x08

（A）中断响应时间是固定的　（B）嵌套中断级别是固定的

（C）中断入口地址是固定的　（D）中断入口地址是可以改变的

（A）中断响应时间是固定的　（B）嵌套中断级别是固定的

（C）中断入口地址是固定的　（D）中断入口地址是可以改变的

（A）将Interrupt子函数的地址赋给T0

（B）将Interrupt子函数的地址赋给T1

（C）将Interrupt子函数的地址赋给T0

（D）将Interrupt子函数的地址赋给T1

第7章 定时器/计数器的应用

本章内容丰富，主要包括以下两部分。

硬件部分：

认识8051定时器/计数器的结构及其4种模式。

认识89S51的看门狗定时器。

程序与实践部分：

定时器的应用程序。

码表程序。

频率产生器程序。

计频器应用程序。

看门狗定时器应用程序。

7.1 8051的定时器/计数器



定时器/计数器（timer/counter）是一种计数装置，若计数内部的时钟脉冲，可视为定时器；若计数外部的脉冲，可视为计数器。而定时器/计数器的应用可以采用中断的方式，就像第6章所介绍的，当定时或计数达到终点时即产生中断，即CPU将暂时放下目前正在执行的程序，先去执行特定的程序，待完成特定的程序后，再返回刚才放下的程序。比如说，老师正在讲课，突然下课铃响了，老师立即暂停课程进度，先下课，等下次上课，再继续刚才暂停的课程。另外，我们也可以采用询问方式，不断询问计数状态，判断程序流程的跳转，稍后也讲到这方面的实例应用。

7.1.1 MCS-51的定时器/计数器

8051提供两个16位的定时器/计数器，分别是Timer 0及Timer 1（简称T0及T1），而8052则提供3个16位的定时器/计数器，除了8051的T0与T1外，还多一个Timer 2。这3个定时器/计数器可作为内部定时器或外部计数器，若当成内部定时器时，则是计数内部的脉冲。以12MHz的计数时钟脉冲系统为例，将此计数时钟脉冲除12后送入定时器，因此，定时器所计数的脉冲周期为1μs。若采用16位的定时模式，则最多可计数2^{16}个脉冲（**65 536**），约0.0655s。

若当成外部计数器时，则是计数由T0或T1引脚送入的脉冲。同样地，若采用16位的定时模式，则最多可计数2^{16}个脉冲，也就是65 536个计数量，相当可观！

MCS-51的定时器/计数器可设计成4种工作模式，分别是Mode 0、Mode 1、Mode 2及Mode 3，如表7-1所示。

表7-1

模式	位数	计数范围	其他功能
Mode 0	13位	0~8191	
Mode 1	16位	0~65535	
Mode 2	8位	0~255	具有自动加载功能
Mode 3	8位	0~255	

应用定时器/计数器时，除了第6章所介绍的IE寄存器和IP寄存器外，还会使用到定时器/计数器模式寄存器（TMOD）、定时器/计数器控制寄存器（TCON）、计数寄存器（THx、TLx）等，这些寄存器将在接下来的章节中介绍。

7.1.2 定时器/计数器模式寄存器TMOD

定时器/计数器模式寄存器TMOD的功能是设定定时器/计数器的工作模式、计数信号来源及启动定时器/计数器方式等，如图7-1所示。

TMOD模式寄存器的高四位（TMOD.7~TMOD.4）用来设定Timer 1的工作模式，而低四位（TMOD.3~TMOD.0）用来设定Timer 0的工作模式。这两部分的结构类似，只是控制对象

不同而已。以低四位为例，GATE位为Timer的门控开关，用来决定其启动方式。若GATE=0，则只要TR0位为1，即可启动Timer 0，称为内部启动或软件启动。若GATE=1，则需要先将TR0位设定为1，再等待 $\overline{\text{INT0}}$ 引脚为高电平，即可启动Timer 0，称为外部启动或硬件启动。C/$\overline{\text{T}}$位为定时器/计数器切换开关，若C/$\overline{\text{T}}$位=0，则Timer 0为内部定时器，用以计数由OSC/12的脉冲。若C/$\overline{\text{T}}$位=1，则Timer 0即为外部计数器，用以计数由T0引脚输入的脉冲。

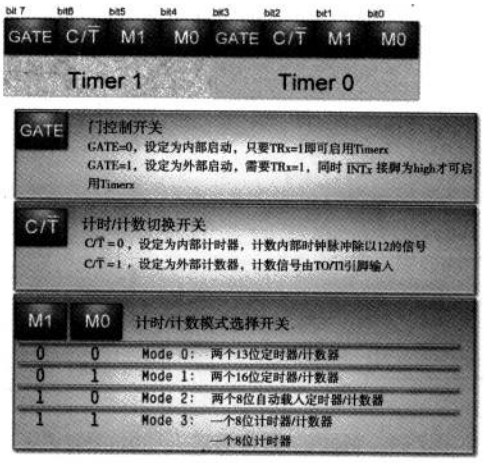


图7-1 TMOD寄存器

M1及M0两个位可设计工作模式，这4种工作模式说明如下。

MODE 0

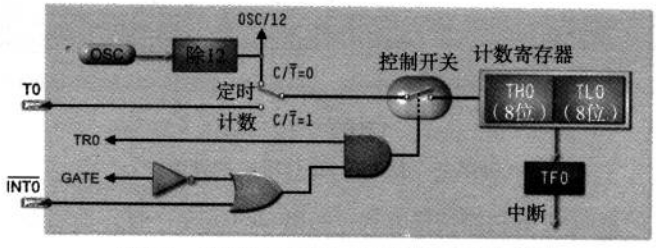


图7-2 MODE 0工作模式（Timer 0为例）

如图7-2所示，Mode 0工作模式提供两个13位的定时器/计数器（Timer 0及Timer 1），其计数量分别放置在TH*x*与TL*x*两个8位的计数寄存器里，其中TH*x*放置8位，TL*x*放置5位，如图7-3所示。

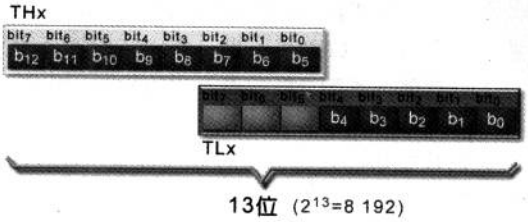


图7-3 Mode 0工作模式的计数

若要执行定时功能，则将 C/$\overline{T}$ 位设定为0，CPU将计数被12除的系统频率，每个频率为1μs。若要执行计数功能，则将 C/$\overline{T}$ 位设定为1，CPU将计数从T*x*引脚输入的脉冲。

定时器/计数器受“控制开关”控制，开启这个开关的方法有两个，第一种是外部启动，也就是将GATE位设定为1，再将TR*x*位设定为1，然后等待 $\overline{\text{INT}x}$ 引脚的信号，当 $\overline{\text{INT}x}$ 引脚为高电平时，即可启动这个定时器/计数器。第二种是内部启动，也就是将GATE位设定为0，接下来，只要将TR*x*位设定为1，即可启动这个定时器/计数器。

Mode 1

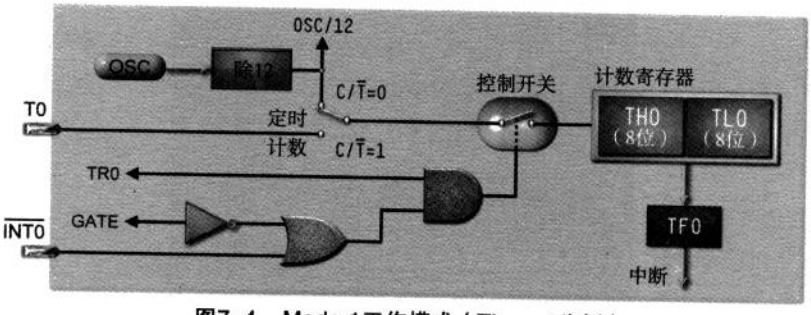


图7-4 Mode 1工作模式（Timer 0为例）

如图7-4所示，Mode 1工作模式提供两个16位的定时器/计数器（Timer 0及Timer 1），其计数量分别放置在TH*x*与TL*x*两个8位的计数寄存器里，其中TH*x*放置8位，TL*x*放置8位，如图7-5所示。

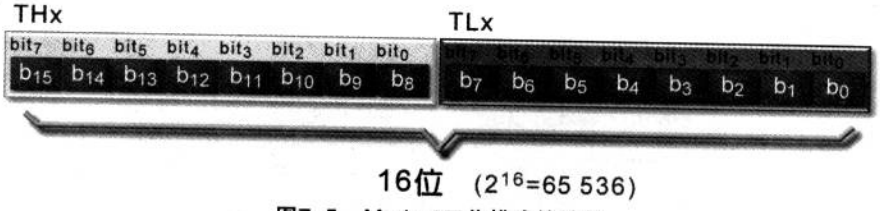


图7-5 Mode 1工作模式的计数

此工作模式的定时/计数功能切换方式，与Mode 0完全一样；而启动定时器/计数器的方式，与Mode 0完全一样。计数量Mode 1又比Mode 0还大！换言之，Mode 1可以完全取代Mode 0，所以，很少人使用Mode 0了（难用又没必要）。

Mode 2

如图7-6所示，Mode 2工作模式提供两个8位可自动加载的定时器/计数器（Timer 0及Timer 1），其计数量放置在TL*x*计数寄存器里，当该定时器/计数器中断时，将会自动将TH*x*计数寄存器里的计数量载入到TL*x*里。由于只有8位，因此，其计数范围仅到256为止。

此工作模式的定时/计数功能切换方式，与Mode 0完全一样；而启动定时器/计数器的方式，也与Mode 0完全一样。

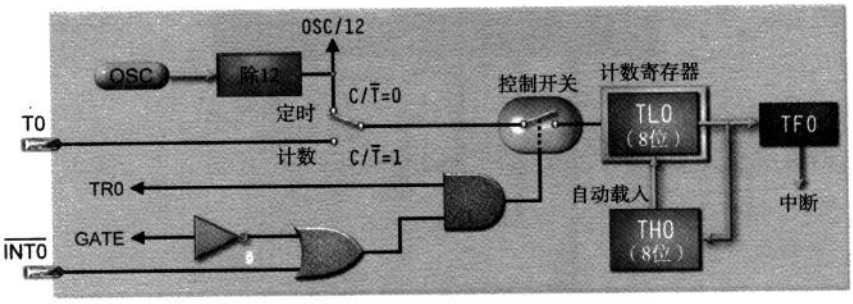


图7-6 MODE 2工作模式（Timer 0为例）

MODE 3

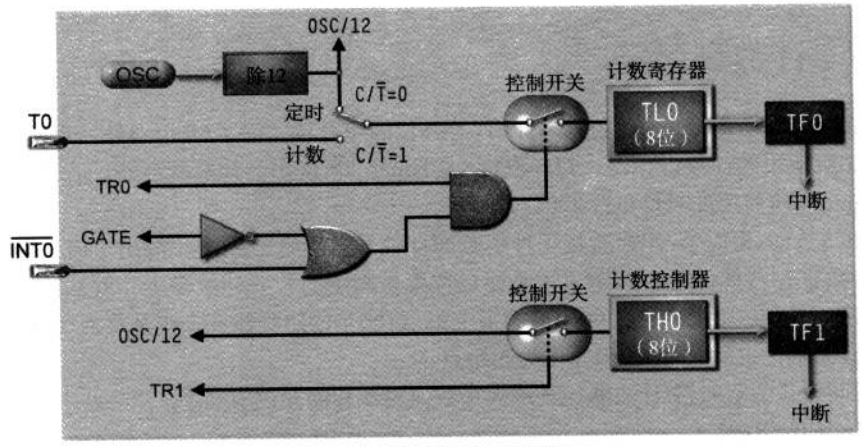


图7-7 MODE 3工作模式

如图7-7所示，Mode 3工作模式是一种特殊的模式，提供一个8位的定时器/计数器Timer 0及一个8位的定时器Timer 1，而其奇特的结构已经不太像真正的Timer 0或Timer 1了。

其中的Timer 0定时器/计数器是由T0、$\overline{\text{INT0}}$引脚、TR0、GATE位以及TL0计数寄存器所构成的，除了不具有自动加载功能外，与Mode 2的Timer 0几乎完全一样。

其中的Timer 1定时器是由TR1位及TH0计数寄存器所构成的，除了不具有计数及自动加载功能外，几乎可以被Mode 2的Timer 1所取代。虽然看不出有任何Mode 3工作模式存在的理由，但这个模式的存在是事实。

7.1.3 定时器/计数器控制寄存器TCON

定时器/计数器控制寄存器TCON的高四位提供定时器/计数器的启动开关以及中断时的标志位，如图7-8所示。

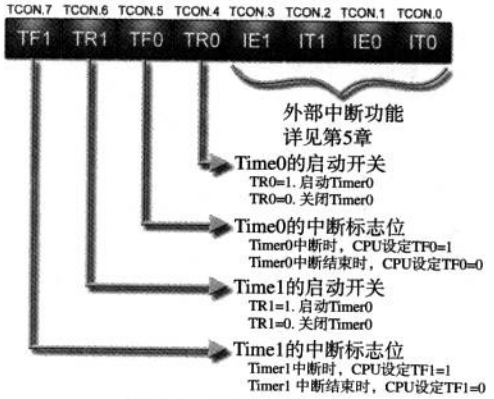


图7-8 TCON寄存器

7.1.4 计数寄存器

MCS-51的计数寄存器是TH*x*及TL*x*两个8位的寄存器，除了Mode 3外，TH0和TL0是Timer 0所使用的计数寄存器，TH1和TL1是Timer 1所使用的计数寄存器。若是8052，则还有Timer 2所使用的TH2和TL2计数寄存器。

MCS-51的定时器/计数器是一种正数的计数器，当计数到满（溢出）时，即产生中断。计数寄存器就像是一条跑道，其终点位置是固定的，要计数多少，就从终点往前推多少以作为起点。例如要在400米的跑道上举行100米的跑步比赛，则从终点（400米）处往前推100米，也就是将300米处作为起跑点。

不同模式的定时器/计数器，其最大计数值各不相同，说明如下。

Mode 0：8192

Mode 1：65536

Mode 2及Mode 3：256

而设定计数量的方式也有所差异，说明如下。

MODE 0

由于在Mode 0工作模式下，TL*x*计数寄存器只使用5位，而2^5=32，我们要把计数起点的值除以32，其余数放入TL*x*计数寄存器；其商放入TH*x*计数寄存器。例如要使用Timer 0计数6000，则填入计数寄存器的指令如下。

```
TL0=(8192-6000)%32;          // 取5位的余数
TH0=(8192-6000)/32;          // 取5位的商
```

MODE 1

由于在Mode 1工作模式下，TL*x*和TH*x*计数寄存器各使用8位，而2^8=256，我们要把计数起点的值除以256，其余数放入TL*x*计数寄存器；而其商放入TH*x*计数寄存器。例如

要使用Timer 0计数50000，则填入计数寄存器的指令如下。

```
TL0=(65536-50000)%256;          // 取8位的余数
TH0=(65536-50000)/256;          // 取8位的商数
```

MODE 2

由于在Mode 2工作模式下，只使用TL*x*计数寄存器，TH*x*计数寄存器作为自动加载的值，而其中都使用8位（2^8=256），所以只要用256减去计数起点的值，再分别放入TL*x*及TH*x*计数寄存器即可。例如要使用Timer 0计数100，则填入计数寄存器的指令如下。

```
TL0=256-100;                    // 填入计数值
TH0=256-100;                    // 填入自动加载值
```

MODE 3

由于在Mode 3工作模式下，使用TL0计数寄存器作为第一个定时器/计数器的计数量，而TH0计数寄存器作为第二个定时器的计数量。用到的定时器/计数器或定时器才需要填入，例如只要使用第一个定时器/计数器，则只需要填入TL0计数寄存器；若只要使用第二个定时器，则只需要填入TH0计数寄存器；若两个都要使用，则分别将个别的值填入TL0及TH0计数寄存器。填入TL0或TH0计数寄存器的方法与Mode 2一样。

7.1.5 定时器/计数器的应用

定时器/计数器有两种应用方式，第一种是中断方式，第二种是查询方式。若采用中断方式，则包括5项措施，即定时器/计数器中断的设定、计数量的设定、启动定时器/计数器以及中断子程序的编写；若采用查询方式，则不需要中断设定，也不需要中断子程序，只要设定计数量及启动定时器/计数器，然后就判断定时器/计数器的标志位（TF*x*）是否动作，以决定程序流程。以中断方式为例，其5项措施说明如下。

中断设定

中断的设定包括开启中断开关（即IE寄存器的设定）、中断优先等级的设定（即IP寄存器的设定）、中断信号的设定（即TCON寄存器的设定）等。IE寄存器及IP寄存器的设定，例如要开启“总开关”、“T0开关”，可以采用如下命令。

```
IE=0x82;        // 开启中断总开关及T0开关
```

同理，IP寄存器和TMOD寄存器的设定也可以采用类似的指令，例如要把T1中断的优先等级提高，并设定为内部定时器、软件启动方式及Mode 1，可以采用如下命令。

```
IP=0x02;        // 提升T0中断的优先等级
TMOD=0x01;      // 设定为内部定时器、软件启动和Mode 1
```

计数量设定

在启动定时器/计数器之前，必须先设定计数量，而设定计数量的方法根据工作模式的不同而有所不同，在7.1.4节已经详细讲解，在此不再赘述。

启动定时器/计数器

启动定时器/计数器的方法，若是采用软件启动，则只要在程序中出现如下命令即可启动。

```
TRx=1;                          // 启动Timer x
```

其中的“x”是指定时器/计数器的代号。例如要启动Timer 0，则使用“TR0=1”命令。若要停用Timer 0，则使用“TR0=0”命令。

中断子程序

定时器/计数器的中断子程序，与第6章所介绍的外部中断的中断子程序类似，中断子程序第一行的格式为：

```
void 中断子程序名称（void）interrupt中断编号 using 寄存器组
```

其中定时器/计数器的中断编号与外部中断的中断编号不一样，Timer 0的中断编号为1，Timer 1的中断编号为3，Timer 2的中断编号为5。例如要定义一个Timer 1的中断子程序，其名称定义为“my_T1”，而在该中断子程序应声明为：

```
void my_T1 (void)interrupt 3
```

接着，在一对大括号里编写此中断子程序的内容。中断子程序与一般函数类似，稍后将对其进行介绍。

7.2 8052的Timer 2



本节为可选内容，若学习时间不够，或不使用8052，则可跳过本节。Timer 2是8052才有的定时器/计数器，其结构与用法和Timer 0和Timer 1有些不同，Timer 2有如下3种工作模式。

7.2.1 T2CON寄存器

T2CON寄存器是Timer 2的控制寄存器，如图7-9所示对其中各位的功能说明如下。

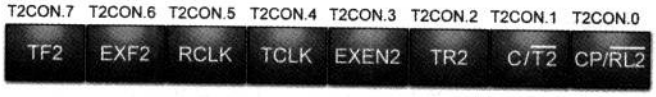


图7-9 T2CON寄存器

- **TF2**：本位为Timer 2溢出标志位，当Timer 2中断时，CPU会将TF2位设定为1；不过，结束Timer 2中断时，CPU并不会将TF2恢复，必须在程序中用“TF2=0;”命令将它恢复为0。
- **EXF2**：本位为Timer 2的外部标志位，当T2EX引脚（即P1.0）输入负边缘信号时，且EXEN2位为1，即进入“捕获模式”或“自动加载模式”，此时EXF2位将被设定为1，并产生Timer 2中断；不过，结束Timer 2中断时，CPU并不会将EXF2恢复，必须在程序中用“EXF2=0;”命令将它恢复为0。
- **RCLK**：本位为串行端口接收频率选择位。当RCLK位为1时，串行口将以Timer 2溢出脉冲作为在Mode 1或Mode 3模式时接收的频率信号。若RCLK位为0，则串行口将以Timer 1溢出脉冲作为接收的频率信号。
- **TCLK**：本位为串行端口传输频率选择位。当TCLK位为1时，串行口将以Timer 2溢出脉冲作为在Mode 1或Mode 3模式时传输的频率信号。若TCLK位为0，则串行口将以Timer 1溢出脉冲作为传输的频率信号。
- **EXEN2**：本位为Timer 2的外部启用控制位，当本位为1时，若Timer 2未被作为串行口的频率产生器，且T2EX引脚输入一个负边缘触发信号，即可使Timer 2进入捕获模式或自动加载模式。若本位为0时，则Timer 2将不理T2EX引脚的信号变化。
- **TR2**：本位为Timer 2的启动位，当本位为1时，即可启动Timer 2。若本位为0时，则停用Timer 2。
- **C/$\overline{\text{T2}}$**：本位为Timer 2定时器/计数器功能切换开关，当本位为1时，Timer 2将执行外部计数功能，以计数T2引脚所输入的脉冲信号。若本位为0时，则Timer 2将执行内部定时功能，以计数系统的时钟脉冲。
- **CP/$\overline{\text{RL2}}$**：本位为Timer 2的工作模式切换位，当本位为1时，若EXEN2=1，且T2EX引脚输入一个负边缘触发信号，Timer 2将产生捕获的动作，将TH2与TL2的数据存入RCAP2H与RCAP2L。当本位为0时，若有溢出发生，或EXEN2=1，且T2EX引脚输入一个负边缘触发信号，Timer 2将产生自动加载的动作，将RCAP2H与RCAP2L的数据加载TH2与TL2。

Timer 2提供3种工作模式，其设定方式可归纳如表7-2所示。

表7-2

RCLK+TCLK	CP/$\overline{\text{RL2}}$	TR2	Mode
0	0	1	16位自动加载模式
0	1	1	16位捕获模式
1	X	1	比特率产生器模式
X	X	0	不使用

7.2.2 捕获模式

捕获模式（Capture Mode）是将TH2与TL2寄存器的数据（16位）输入RCAP2H及RCAP2L寄存器，其结构如图7-10所示。

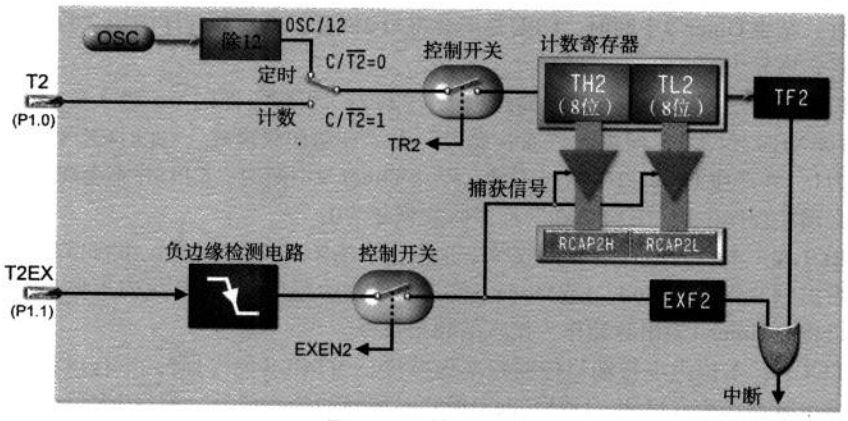


图7-10 捕获模式

若要使用捕获模式，必须将T2CON寄存器里的CP/$\overline{RL2}$位设定为1。如同Timer 0、Timer 1一样，捕获模式可计数内部时钟脉冲（OSC/12），也可以计数由T2引脚输入的外部脉冲。只要将T2CON寄存器里的C/$\overline{T2}$位设定为0，则为内部定时器；将T2CON寄存器里的C/$\overline{T2}$位设定为1，则为外部计数器。另外，T2CON寄存器里的EXEN2位也要设定为1，才能进行捕获模式。而Timer 2的启动开关为TR2，若将TR2设定为1，即可启动Timer 2；TR2=0，即可停用Timer 2。

启动Timer 2后，Timer 2即进行计数工作。若检测到T2EX引脚输入信号中含有负边缘信号，即启动捕获信号，将当时TH2寄存器的内容将被复制到RCAP2H寄存器；TL2寄存器的内容将被复制到RCAP2L寄存器，同时EXF2位设定为1，并产生Timer 2中断。不过，Timer 2的中断并不影响计数的动作，待Timer 2计数溢出时，则TF2位设定为1，并产生Timer 2中断。

上述的描述归纳起来就是，若要采用捕获模式工作，必须进行以下设定。

1. CP/$\overline{RL2}$=1
2. EXEN2=1

再使TR2=1，即可进入捕获模式，Timer 2即可计数。若T2EX引脚输入信号中含有负边缘，即启动捕获信号，同时产生Timer 2中断。当Timer 2计数溢出，又产生Timer 2中断。

7.2.3 自动加载模式

自动加载模式（Auto-Reload Mode）是自动将RCAP2H及RCAP2L寄存器的数据（16位），加载TH2与TL2寄存器，其结构如图7-11所示。

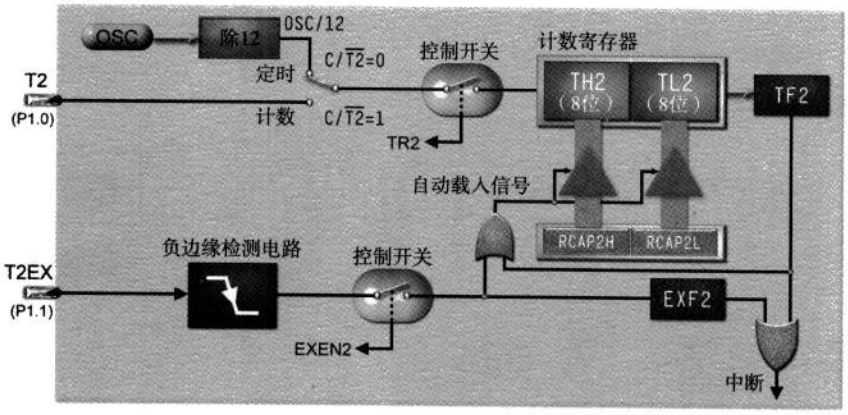


图7-11 自动加载模式

若要使用自动加载模式，必须将T2CON寄存器里的CP/$\overline{RL2}$位设定为0。Timer 2的自动加载模式与Timer 0、Timer 1的Mode2类似，只有Timer 0、Timer 1的Mode2是8位的自动加载功能，Timer 2的自动加载模式则是16位。同样地，自动加载模式可计数内部时钟脉冲（OSC/12），也可以计数由T2引脚输入的外部脉冲。只要将T2CON寄存器里的C/$\overline{T2}$位设定为0，则为内部定时器；将T2CON寄存器里的C/$\overline{T2}$位设定为1，则为外部计数器。另外，T2CON寄存器里的EXEN2位也要设定为1，才能进行自动加载模式。而Timer 2的启动开关为TR2，若将TR2设定为1，即可启动Timer 2；TR2=0，即可停用Timer 2。

启动Timer 2后，Timer 2即进行计数工作，若检测到T2EX引脚输入信号中含有负边缘，即启动自动加载信号，将当时RCAP2H寄存器的内容将被复制到TH2寄存器；RCAP2L寄存器的内容将被复制到TL2寄存器，同时EXF2位设定为1，并产生Timer 2中断。不过，Timer 2的中断并不影响计数的动作，待Timer 2计数溢出时，则TF2位设定为1，并产生Timer 2中断。

将上述的描述归纳起来，若要采用自动加载模式工作，必须进行如下设置：

1. CP/$\overline{RL2}$=0
2. EXEN2=1

再使TR2=1，即可进入自动加载模式，Timer 2即可计数。若T2EX引脚输入信号中含有负边缘，即启动自动加载信号，同时产生Timer 2中断。当Timer 2计数溢出，又产生Timer 2中断。

7.2.4 比特率产生器模式

MCS-51的串行口传输率（比特率）可由Timer 1或Timer 2所产生的溢出脉冲来控制，Timer 2的比特率产生器模式（Baud Rate Generator Mode）就是提供串行口传送与接收的时钟脉冲，其结构如图7-12所示。

如图7-12所示，在比特率产生器模式下，Timer 2可分为两个独立的部分，在下面的区块里，若EXEN2位设定为1，则只要检测到T2EX引脚上有负边缘信号，即可使EXF2标志位设定为1，同时产生Timer 2中断。

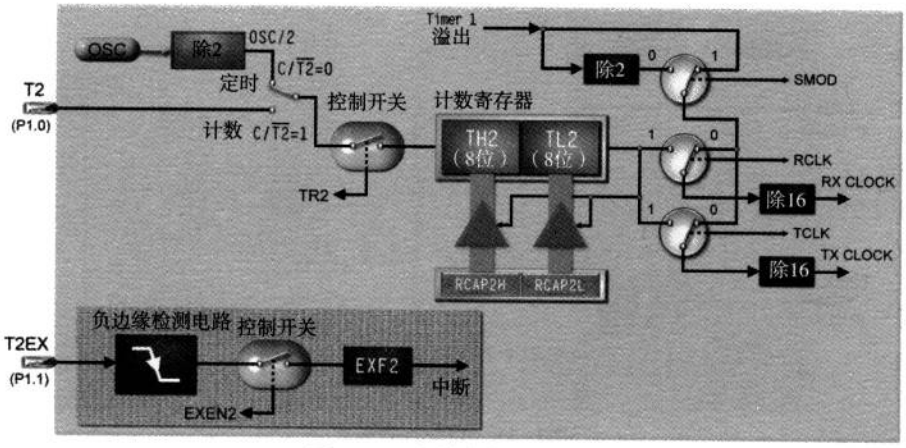


图7-12　比特率产生器模式

在其他部分则为比特率产生器，若要使用Timer 2比特率产生器所产生的时钟脉冲，则需要T2CON寄存器里的RCLK位或TCLK位设定为1。当RCLK=1，则Timer 2比特率产生器将提供串行口接收所需的时钟脉冲；TCLK=1，则Timer 2比特率产生器将提供串行口传送所需的时钟脉冲。此外，T2CON寄存器里的 $C/\overline{T2}$ 位设定为1，Timer 2将计数由T2引脚所输入的外部脉冲信号，以产生溢出脉冲；若 $C/\overline{T2}$ 位设定为0，Timer 2将计数由OSC/2的内部时钟脉冲信号，以产生溢出脉冲，因此，其所产生的比特率为：

$$\frac{\text{OSC/2}}{16\times[65536-(\text{RCAP2H},\text{RCAP2L})]}$$

当然，Timer 2的启动开关还是为TR2，若将TR2设定为1，即可启动Timer 2；TR2=0，即可停用Timer 2。若RCLK位或TCLK位都不为1，则串行口将采用Timer 1所产生的时钟脉冲。

7.3 8051的省电模式

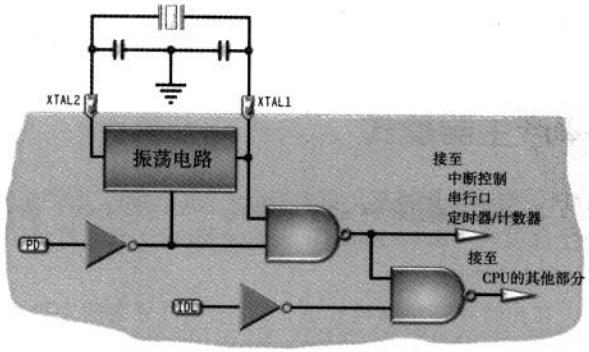


图7-13　8051功率控制示意图

顾名思义，“省电模式”就是要让系统的耗电量降低，同时又能保存系统中的数据，如此才能在使用电池的状态下长时间运作。8051的CHMOS版本提供两种省电模式，即闲置模式（idle mode，简称IDL模式）与功率下降模式（power-down mode，简称PD模式）。如图7-13所示为8051内部的功率控制示意图，其中的IDL端点与PD端点连接到电源控制寄存器PCON的IDL及PD位，而此部分控制了整个CPU所需的时钟脉冲，说明如下。

7.3.1 闲置模式

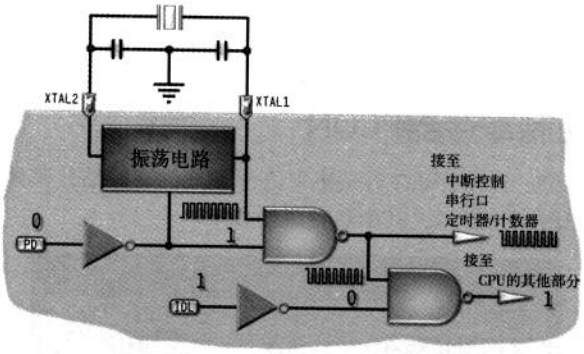


图7-14 闲置模式

若IDL端点为1，则进入闲置模式，除了中断、串行口、定时器/计数器等仍正常提供时钟脉冲外，CPU的其他部分均无时钟脉冲。因此，CPU将停止运行，但其中各寄存器、堆栈、内存、输入/输出端口等的数据并不会消失。如图7-14所示为闲置模式的状态，若要结束闲置模式，只要让IDL端点为0，即可正常提供各部分时钟脉冲，CPU将恢复正常运作。要让IDL端点为0，可以使用下面两种方法。

1. 启动任意一个中断启用。

2. 让系统复位，也就是让RESET引脚（第9脚）为高电平，持续2个机器周期（2μs），则CPU内各寄存器恢复为初始状态，PCON寄存器里的IDL位将恢复为0，也就是说IDL端点为0。不过，系统复位后，各寄存器、输入/输出端口等的数据将消失，但内存内的数据仍在。所以，除非万不得已不要这样做。

7.3.2 功率下降模式

若PD端点为1，则进入功率下降模式，此时，完全不提供时钟脉冲，功率损耗降至最低。外加电源也可由原本的+5V降至+2V。当然，各寄存器、堆栈、内存、输入/输出端口等的数据并不会消失。如图7-15所示为功率下降模式的状态。

若要结束功率下降模式，必须先将电源恢复+5V，然后让系统复位，即让RESET引脚（第9脚）为高电平，且需要持续10ms以上。

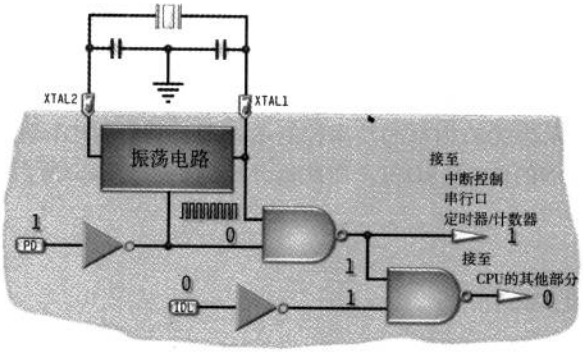


图7-15 功率下降模式

7.3.3 电源控制寄存器PCON

由上述介绍可得知，PCON寄存器是主宰电源管理的寄存器，其地址为87H，不是可位寻址的寄存器，如图7-16所示，其中各位说明如下。

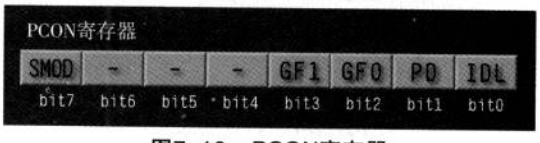


图7-16 PCON寄存器

- SMOD位为比特率（baud rate）倍增位。当串行端口工作于模式1、模式2和模式3，且使用定时器1为其比特率产生器时，若SMOD位设定为1，则比特率加倍；若SMOD位设定为0，则比特率正常。
- GF1与GF0位为一般用途标志位，使用者可自行设定或清除这两个标志位。通常，我们是使用这两个标志位作为由中断唤醒闲置模式中的8051系统。
- PD位为功率下降模式位。当PD=1，即可进入功率下降模式；PD=0，即可结束功率下降模式。
- IDL位为闲置模式位。当IDL=1，即可进入闲置模式；IDL=0，即可结束闲置模式。

当系统复位时，PCON寄存器的初始状态恢复为0XXX0000B（CHMOS版本的8051），若是HMOS版本的8051，则为0XXXXXXXB。

7.4 认识看门狗定时器



看门狗定时器（Watchdog Timer，简称为**WDT**）是一种微处理机防呆装置，当系统超过某个时间没有动作时，WDT就自动复位（RESET），让系统复归为正常运作状态。大部分微处理机都内含WDT，51系列早内部就有WDT的特殊规格芯片，而到了89S51才正式列为基本配置。

顾名思义，“看门狗”就是要来帮微处理机看家的小宠物。在第1章中介绍8051的引脚时，曾经提及“系统久久不动，就要按一下RESET按钮，以使系统复位”，这“系统久久不动”就是俗称的死机！哪有微处理机不死机的？不管是什么因素引起的死机，死机后，微处理机工作必然不正常。看门狗定时器是个很有意思的东西，更是一个不甘寂寞的装置，若系统太久没有来复位看门狗定时器，看门狗定时器就会使系统复位！因此，在执行程序时，必须在看门狗定时器还没到复位系统之前，就先重新启动看门狗定时器，让它没有机会使系统复位。万一系统死机或不正常，系统就无法实时重新启动看门狗定时器，看门狗定时器就会使系统复位。

89S51内含的看门狗定时器是由一个**14**位定时器及**WDTRST**（即**Watchdog Timer Reset**）寄存器所构成的，而WDT的预置状态是停用（disable）。若要启用（enable）WDT功能，则需要依次将0x1e和0xe1放入WDTRST寄存器，此寄存器的位置是0xa6。启用WDT后，此计数器将随时钟脉冲的机器周期递增计数（一个机器周期，WDT加1）。当启用WDT后，只有复位（不管是直接由RESET引脚复位，还是WDT溢出的复位）才会停用WDT。虽然启用WDT后不可停用，但可以复位WDT，让它重新定时；而复位WDT的方法与启用WDT一样，只要依次将0x1e和0xe1放入WDTRST寄存器即可。换言之，程序必须在WDT溢出之前将0x1e和0xe1放入WDTRST寄存器的动作，否则，CPU将被复位，程序就不能顺利执行。

当14位定时器溢出（达到16383，即0x3fff），即由RESET引脚送出一个高电平脉冲以复位装置。每16384（即2^{14}）个机器周期（溢出），WDT将产生一个高电平的复位信号；换言之，以12MHz的时钟脉冲为例，每16384μs（约0.016s）即产生一个复位信号。而此复位信号的脉冲宽度为$98 \times TOSC$，其中TOSC=1/FOSC，脉冲的宽度为$Width=98 \times \frac{1}{12M} \cong 8.167\mu s$。

启用WDT与复位WDT

当我们要启用WDT或复位WDT时，可在汇编语言的程序中加入下列命令。

```
mov     WDTRST, #1eh
mov     WDTRST, #e1h
```

若要在C语言里启用WDT或复位WDT功能，由于reg51.h之中并没有声明WDTRST寄存器，所以我们必须先声明WDTRST寄存器，命令如下。

```
sfr     WDTRST = 0xa6;
```

进行了如上的声明以后，即可在程序中以下列命令启用WDT或复位WDT功能。

```
WDTRST = 0x1e;
WDTRST = 0xe1;
```

在功率下降模式下的WDT

在功率下降模式下，内部振荡器将停止，当然WDT也就停止运作。因此，不必进行WDT溢出之前复位WDT的动作。不过，在进行退出功率下降模式时，WDT就可能会有影响。退出功率下降模式的方法有两个，第一个方法是由硬件复位，第二个方法是由低电平驱动外部中断，在前面的章节中已说明过了。

若以硬件复位方式退出功率下降模式，那就和平时一样，硬件复位之后，程序必须在WDT溢出之前复位WDT，以防止WDT将CPU复位。

若以外部中断的方式退出功率下降模式，情况就大不相同！由于以外部中断时，外部中断引脚必须为低电平，且持续一段时间，让系统的振荡电路趋于稳定。所幸外部中断引脚为低电平时，WDT并不启动，直到外部中断引脚为高电平时，才会启动。

不管使用哪种退出功率下降模式的方式，在此建议，进入功率下降模式之前，最好能复位WDT，以确保在外部中断的方式退出功率下降模式时，WDT不会在系统恢复正常不久，即溢出而复位CPU。

在闲置模式下的WDT

在AUXR寄存器中，WDIDLE位的功能是用来决定WDT在闲置模式下是否继续计数。若WDIDLE=0（预置状态），则在闲置模式下，WDT仍然继续计数。为了防止WDT在闲置模式模式时复位CPU，我们可周期性的退出闲置模式模式、复位WDT，再重新进入闲置模式模式。

当启用WDIDLE位（即WDIDLE=1）时，在闲置模式下WDT将停止计数，直到退出闲置模式后，WDT才会恢复计数。

在reg51.h之中也没有声明AUXR寄存器及WDIDLE位，所以我们必须先声明它，使用如下命令。

```
sfr                                    AUXR= 0xa2;
```

即可在程序之中使用如下命令，让CPU在闲置模式时，WDT停止计数。

```
AUXR = 0x10;
```

7.5 实例演练



在本单元里提供4个范例，如下所示。

7.5.1 闪烁灯实例演练-查询方式

实验要点

本实验的目的只是为了解8051的Timer的应用，所以利用非常简单的电路与程序，让

大家知道如何用C语言设计Timer中断程序。如图7-17所示，由P2驱动的8个LED，我们将设计一个程序，每0.25s这8个LED交互闪烁一次。而Timer的应用可分为查询方式与中断方式，所谓“查询方式”是主程序什么也不做，只是不断地查询Timer是否中断了，而不需要准备中断子程序。“中断方式”则是主程序专注于其他事情，待Timer中断时，才执行中断子程序。在此将以查询方式进行闪烁灯的控制。

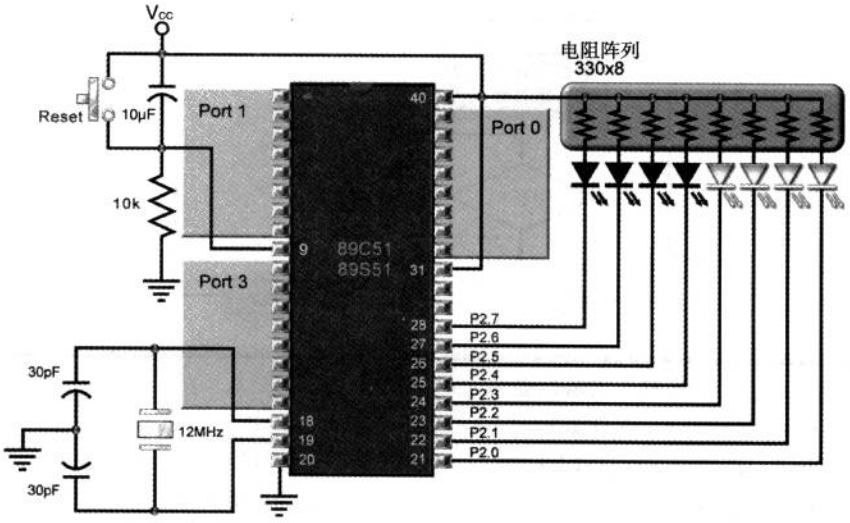


图7-17 闪烁灯实验电路图

流程图与程序设计

在此可使用Timer 0或Timer 1来定时，而不管使用哪个Timer，都可使用mode 0、mode 1、mode 2或mode 3。当然，能用mode 0的场合，必然可使用mode 1代替，而可使用mode 3的场合，也必然可使用mode 2代替。因此，只要会应用mode 1与mode 2就够了！对与12MHz的系统说明如下。

1. 当在Mode 0模式工作时，每次最多可计数8 192，约8ms，若只计数**5 000**，则为5ms，必须重复**50**次才延迟0.25s（5m × 50）。

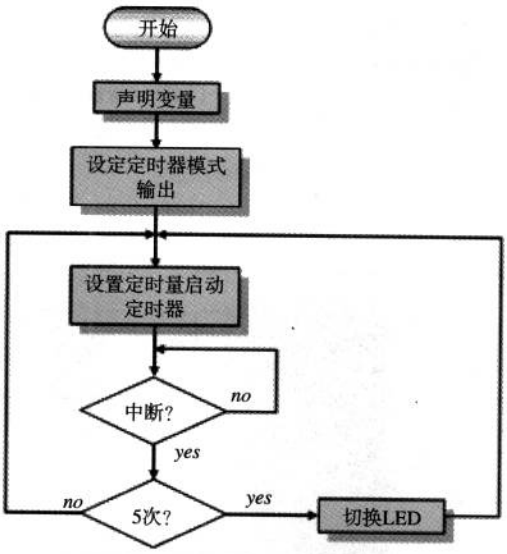
2. 当在Mode 1模式工作时，每次最多可计数65 536，约65ms，若只计数**50 000**，则为50ms，必须重复**5**次才延迟0.25s（50m × 5）。

3. 当在Mode 2或Mode 3模式工作时，每次最多可计数256，约0.25ms，若只计数**250**，则为0.25ms，必须重复**1 000**次才延迟0.25ss（0.25m × 1000）。

在此以Mode 1为例，每次计数量为50000，重复5次后，即将切换LED的状态。不过，查询的设定必须注意以下两点。

1. 不要设定中断启用寄存器，即不开启中断总开关与定时器开关。

2. 当定时器标志位变为**1**之后，还得使用“**TF0=0;**”将定时器标志位变为**0**，该定时器才能重新启用。




```
/* ch7_1.c – 定时器实验I–查询方式–高低位灯互闪 */
//==声明区==============================
#include      <reg51.h>                  // 定义8051寄存器的头文件
#define LED   P2                         // 定义LED接至Port 2
#define count 50000                      // T0(MODE 1)的计数值，约0.05s
#define       TH_M1(65636-count)/256     // T0(Mode 1)计数高8位
#define       TL_M1(65636-count)%256     // T0(Mode 1)计数低8位
//==主程序==============================================
main()                                   // 主程序开始
{   int i;                               // 声明i变量
    IE=0x82;                             // 启用TF0中断
    TMOD=0x01;                           // 设定T0为mode 1
    LED=0xf0;                            // LED初值=1111 0000，右4灯亮
    while(1)                             // 无穷循环，程序一直运行
    {   for (i=0;i<5;i++)                // for循环，定时中断5次
        {       TH0=TH_M1;               // 设置高8位
                TL0=TL_M1;               // 设置低8位
                TR0=1;                   // 启动T0
                while(TF0==0);           // 等待中断(TF0==1)
                TF0=0;                   // 中断后，清除TF0，关闭T0
        }                                // for循环定时结束
        LED=~LED;                        // 输出反相
    }                                    // while循环结束
```

```
}                                    // 主程序结束
```

定时器实验（ch7-1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图7-17连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，所采用的是Timer 0，若要采用Timer 1，应如何修改？

2. 若要使用mode 2来完成本实验的功能，程序应如何修改？

7.5.2 闪烁灯实例演练-中断方式

实验要点

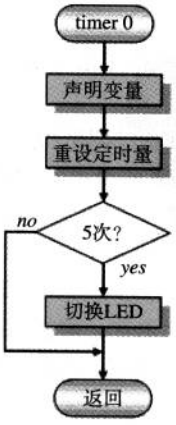
本实验的目的与7.4.1节一样，但在本实验里将采用中断的方式进行。

流程图与程序设计

在此的定时方式与7.4.1节一样，但程序的结构不同，且需要中断子程序，流程图与整个程序如下所示。



主程序



```
/* ch7_2.c - 定时器实验2-中断方式-高低位灯互闪 */
//==声明区==============================
#include<reg51.h>                    // 定义8051寄存器的头文件，P2-17~19
#define LED    P2                    // 定义LED接至Port 2
#define count  50000                 // T0(MODE 1)的计数值，约0.05s
#define        TH_M1(65636-count)/256        // T0(Mode 1)计数高8位
#define        TL_M1(65636-count)%256        // T0(Mode 1)计数低8位
int IntCount=0;                      // 声明IntCount变量，计算T0中断次数
//==主程序==============================
main()                               // 主程序开始
{   IE=0x82;                         // 启用TF0中断
    TMOD=0x01;                       // 设定T0为mode 1
    TH0=TH_M1; TL0=TL_M1;// 设置T0计数量高8位、低8位
    TR0=1;                           // 启动T0
    LED=0xf0;                        // LED初值=1111 0000，右4灯亮
    while(1);                        // 无穷循环，程序停滞
}                                    // 主程序结束
//== T0中断子程序- 每中断5次，LED反相 ================
void timer0(void)interrupt 1         // T0中断子程序开始
{   TH0=TH_M1; TL0=TL_M1; // 设置T0计数量高8位、低8位
    if(++IntCount==5)                // 若T0已中断5次数
    {   IntCount=0;                  // 重新计次
        LED^=0xff;                   // 输出相反
    }                                // if语句结束
}                                    // T0中断子程序
```

定时器实验（ch7-2.c）

● 操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图7-17连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

思考一下

在本实验里，主程序没有做任何事情，只是在等待。若要让主程序进行Port 0的单灯左移，应如何修改？

7.5.3 60s定时器实例演练

实验要点

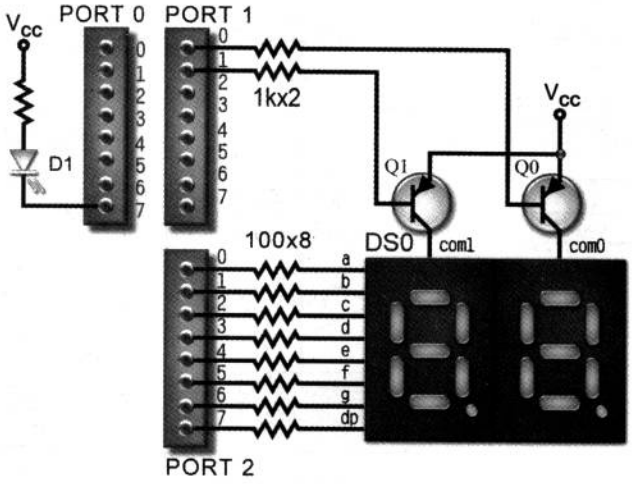


图7-18 60s定时电路图

如图7-18所示，P2驱动两位数7段LED数码管模块，而P1.0与P1.1为两位数7段LED数码管模块的扫描信号，其中P1.0为个位数的扫描信号，P1.1为十位数的扫描信号。在此将利用Timer 0作为定时装置，两个7段LED数码管从“00”开始显示，每1s增加1，到达“59”后，再从“00”开始，也就是60s的定时器。每过60s，D1切换一次（原本亮的，变成灭；原本不亮的，变成亮）。

流程图与程序设计

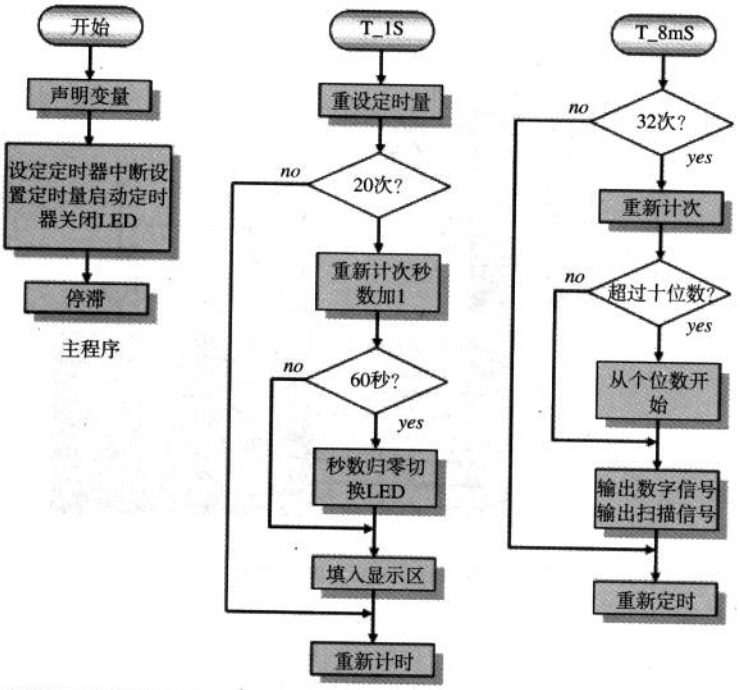
在此要使用Timer 0来定时，每s显示数字加1，若使用mode 1，则计数量设定为50 000（即50ms），重复次数为20，在此设定下列变量：

```
int TH_M1(65636-50000)/256;          // 声明T0计数量的高8位
int TL_M1=(65636-50000)%256;         // 声明T0计数量的低8位
char count0=20;                      // 重复次数
```

而Timer 1作为扫描周期的控制，每8ms扫描一个数字，若也使用mode 1，则只要计数量为8 000，而不需要重复执行。不过，在此刻意使用mode 2，则只要计数量为250，而重复执行32次才为8ms，在此设定下列变量。

```
int TH_M2=(256-250);                    // 声明T1自动加载计数量
int TL_M2=(256-250);                    // 声明T1计数量
char count1=32;                         // 重复次数
```

流程图及整个程序如下所示。




```
/* ch7_3.c - 定时器实验3 -60s计数器，每1分钟LED反相1次 */
//==声明区==================================
#include        <reg51.h>                 // 定义8051寄存器的头文件
#define SEG     P2                        // 定义7段LED数码管接至Port 2
#define SCANP P1                          // 定义扫描线接至Port 1
sbit LED=P0^7;                            // 声明LED接至P0.7
/*声明T0定时相关声明*/                    // THx TLx
#define count_M1   50000                  // T0(MODE 1)的计数值，0.05s
#define TH_M1 (65636-count_M1)/256        // T0(MODE 1)计数高8位
#define TL_M1 (65636-count_M1)%256// T0(MODE 1)计数低8位
int count_T0=0;                           // 计算 T0中断次数
/*声明T1扫描相关声明*/
#define count_M2   250                    // T1(MODE 2)的计数值，0.25ms
#define TH_M2 (256-count_M2)              // T1(MODE 2)自动加载计数
```

```
#define  TL_M2 (256-count_M2)          // T1(MODE 2)计数量
char count_T1=0;                       // 计算T1中断次数
/* 声明7段LED数码管驱动信号数组(共阳)*/
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,    // 数字0~4
               0x92, 0x83, 0xf8, 0x80, 0x98 };  // 数字5~9
char disp[2]={ 0xc0, 0xc0 };           // 声明显示区数组初始显示00
/* 声明基本变量 */
char seconds=0;                        // 秒数
char scan=0;                           // 扫描信号
//==主程序==================================
main()                                 // 主程序开始
{   IE=0x8a;                           // 1000 1010，启用TF0、TF1中断
    TMOD=0x21;                         //00100001，T1采用mode2、T0采用mode1
    TH0=TH_M1; TL0=TL_M1;              // 设置T0计数量高8位、低8位
    TR0=1;                             // 启动T0
    TH1=TH_M2; TL1=TL_M2;              // 设置T1自动加载值、计数量
    TR1=1;                             // 启动T1
    LED=1;                             // 关闭LED
    while(1);                          // 无穷循环，程序停滞
}                                      // 主程序结束
//== T0中断子程序-计算并显示秒数 ==================
void T0_1s(void)interrupt 1            // T0中断子程序开始
{   TH0=TH_M1; TL0=TL_M1;              // 设置T0计数量高8位、低8位
    if(++count_T0==20)                 // 若中断20次，即0.05x20=1s
    {   count_T0=0;                    // 重新计次
        seconds++;                     // 秒数加1
        if(seconds==60)                // 若超过60s
        {       seconds=0;             // 秒数归0，重新开始
                LED=~LED;              // 切换LED
        }                              // if语句结束(超过60s)
    }                                  // if语句结束(中断20次)
    disp[1]=TAB[seconds/10];           // 填入十位数显示区
    disp[0]=TAB[seconds%10];           // 填入个位数显示区
}                                      // T0中断子程序结束
//===T1中断子程序-扫描 ========================
void T1_8ms(void)interrupt 3           // T1中断子程序开始
{   if(++count_T1==32)                 // 若中断32次，即0.25mx32=8ms
        {       count_T1=0;            // 重新计次
```

```
            if(++scan==3)scan=1;          // 若超过十位数，显示个位
            SEG=0xff;                     // 关闭7段LED数码管
            SCANP=~scan;                  // 输出扫描信号
            SEG=disp[scan-1];             // 输出显示信号
        }                                 // 结束if判断(中断32次)
    }                                     // T0中断子程序结束
```

定时器实验（ch7-3.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图7-18连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，所采用的是Timer 0，若要采用Timer 1，应如何修改？

2. 若要使用Mode 2或Mode 3来完成本实验的功能，程序应如何修改？

7.5.4 码表实例演练

实验要点

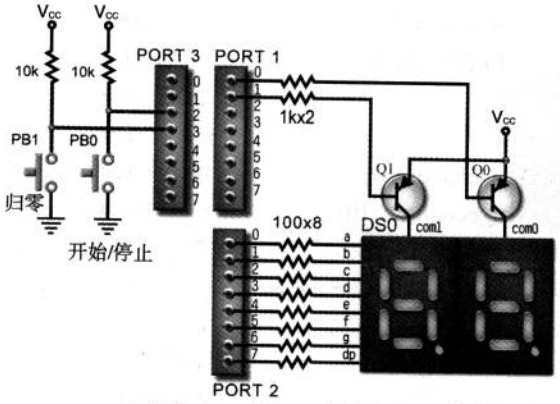
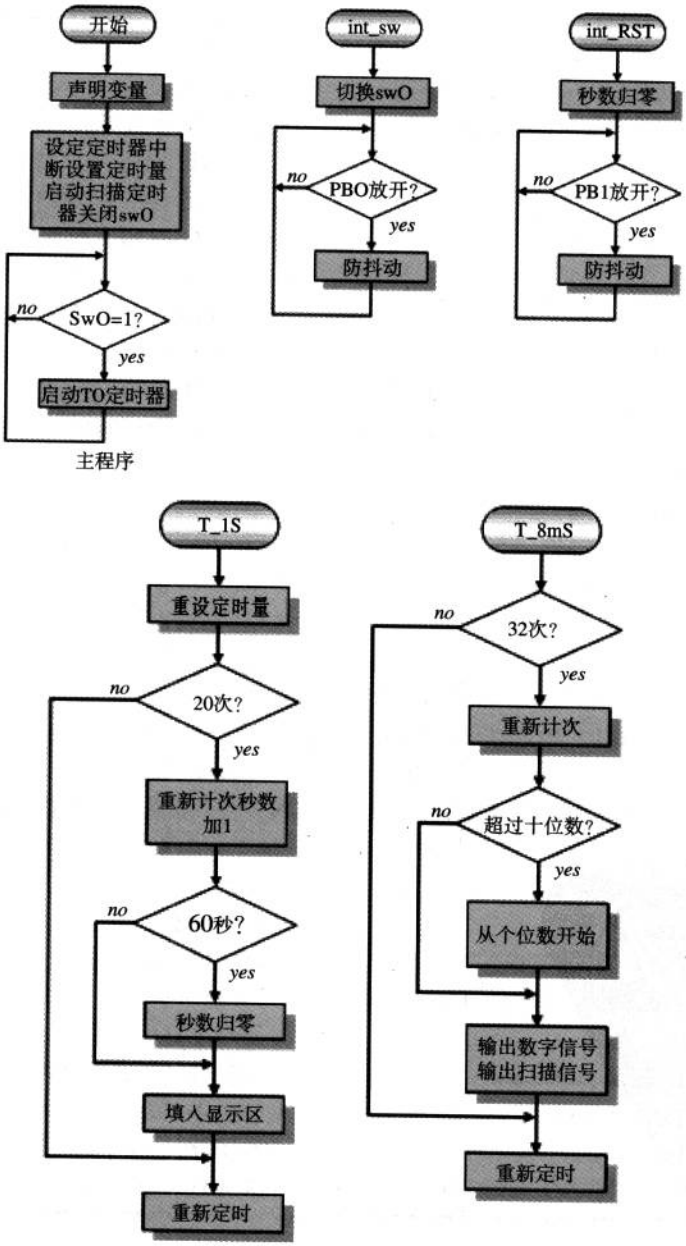


图7-19 简易码表电路图

如图7-19所示，PB0所接的按钮开关具有启动码表及停止码表的功能，按一下PB0按钮开关，即可开始定时。同样地，7段LED数码管上每秒增加1；再按一下PB0按钮开关，即可停止定时。PB1所接的按钮开关的功能是将码表归零，按一下PB1按钮开关，则不管有没有定时，7段LED数码管将从00开始。

流程图与程序设计



基本上，在此所采用的1秒钟定时与显示与7.4.3节类似。而在本单元最主要的是设计一个具有切换功能（Toggle）的中断，也就是第一次PB0中断时，开始定时；第二次PB0中断时，停止定时；第三次PB0中断时，又开始定时……，就像一个切换开关一样。为达此目的，在此利用一个位变量（即sw0），每次PB0中断时，就改变其状态。而1s钟定时与显示里，每次Timer 0中断时，就检查一次sw0，若**sw0=1**，则继续定时与显示；若**sw0=0**，则停止定时。至于PB1中断所要做的事，只是将秒数seconds变成0而已。

```
/* ch7_4.c-码表实验-PB0:开始/暂停 PB1:归零 */
//==声明区==============================
#include      <reg51.h>                  // 定义8051寄存器的头文件
#define SEG   P2                         // 定义7段LED数码管接至Port 2
#define SCANP P1                         // 定义扫描线接至Port 1
sbit    PB0=P3^2;                        // PB0按钮接至P3.2(INT0)
sbit    PB1=P3^3;                        // PB1按钮接至P3.3(INT1)
/*声明T0定时相关声明*/                    // THx TLx
#define count_M1  50000                  // T0(MODE 1)的计数值，0.05s
#define TH_M1  (65636-count_M1)/256 // T0(MODE 1)计数高8位
#define TL_M1  (65636-count_M1)%256      // T0(MODE 1)计数低8位
int count_T0=0;                          // 计算 T0中断次数
/*声明T1扫描相关声明*/
#define count_M2  250                    // T1(MODE 2)的计数值，0.25ms
#define TH_M2  (256-count_M2)            // T1(MODE 2)自动加载计数
#define TL_M2  (256-count_M2)            // T1(MODE 2)计数量
char count_T1=0;                         // 计算 T1中断次数
/* 声明7段LED数码管驱动信号数组(共阳)*/
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,    // 数字0~4
               0x92, 0x83, 0xf8, 0x80, 0x98 }; // 数字5~9
char disp[2]={ 0xc0, 0xc0 };             // 声明显示区数组初始显示00
/* 声明基本变量 */
char seconds=0;                          // 秒数
char scan=0;                             // 扫描信号
void debouncer(void);                    // 声明防抖动函数
//==主程序==============================
main()                                   // 主程序开始
{   IE=0x8f;                             // 10001111，启用INT0/1、TF0/1中断
    TCON=0x00;                           // 设定INT0/1 采用低电平动作
    TMOD=0x21;                           //0010 0001，T1采用mode2、T0采用mode1
    TH0=TH_M1; TL0=TL_M1;                // 设置T0计数量高8位、低8位
```

```
    TR0=0;                                  // 不启动T0
    TH1=TH_M2; TL1=TL_M2;                   // 设置T1自动加载值、计数量
    TR1=1;                                  // 启动T1
    P3=0xff;                                // 设计PB0/PB1输入
    while(1);                               // 无穷循环，程序停滞
}                                           // 主程序结束
//== T0中断子程序-计算并显示秒数 ==================
void T0_1s(void)interrupt 1                 // T0中断子程序开始
{   TH0=TH_M1; TL0=TL_M1;                   // 设置T0计数量高8位、低8位
    if(++count_T0==20)                      // 若中断20次，即0.05x20=1s
    {   count_T0=0;                         // 重新计次
        seconds++;                          // 秒数加1
        if(seconds==60)                     // 若超过60s
        seconds=0;                          // 秒数归0，重新开始
    }                                       // if语句结束(中断20次)
    disp[1]=TAB[seconds/10];                // 填入十位数显示区
    disp[0]=TAB[seconds%10];                // 填入个位数显示区
}                                           // T0中断子程序结束
//===T1中断子程序-扫描 ============================
void T1_8ms(void)interrupt 3                // T1中断子程序开始
{   if(++count_T1==32)                      // 若中断32次，即0.25mx32=8ms
    {   count_T1=0;                         // 重新计次
        if(++scan==3)scan=1;                // 若超过十位数，显示个位
        SEG=0xff;                           // 关闭7段LED数码管
        SCANP=~scan;                        // 输出扫描信号
        SEG=disp[scan-1];                   // 输出显示信号
    }                                       // 结束if判断(中断32次)
}                                           // T0中断子程序结束
//==int0中断子程序-码表的 开始/暂停 ================
void int0_sw(void)interrupt 0               // int0中断子程序开始
{   TR0=~TR0;                               // 切换T0为开始/暂停
    while(PB0==0);                          // 等待放开PB0
    debouncer();                            // 防抖动
}                                           // int 0中断子程序结束
//==int 1中断子程序-码表归零 ======================
void int1_RST(void)interrupt 2              // int 1中断子程序开始
{   while(PB1==0);                          // 等待放开PB1
    debouncer();                            // 防抖动
```

```
        econds=0;                                   // 秒数归零
}                                                   // int 1中断子程序结束
//===防抖动函数=========================================
void debouncer(void)                                // 防抖动函数开始
{    int i;                                         // 声明变量i
     for(i=0;i<2400;i++);                           // 连数2400次，约20ms
}                                                   // 防抖动函数结束
```

码表实验（ch7-4.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图7-19连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里所设计的是0~59s的码表，请将定时范围改成0~99s？

2. 在本实验里使用两位数的7段LED数码管模块，请修改电路，采用四位数的7段LED数码管模块，而其定时范围改成0~9999s。

3. 接续第3题，而其定时范围改成0~59min59s，而这四位数，右边两位为秒数，可从0~59；左边两位为分数，可从0~59，第60s时分数加1，而第60分时归零？

7.5.5 频率发生器实例演练

实验要点

如图7-20所示，指拨开关的状态由P2输入，在此将利用其中的S1~S8设定输出的频率。若S1开关on的话，不管其他开关是什么，OUTPUT端输出100k Hz；若S1开关off，S2开关on的话，不管其他开关是什么，OUTPUT端输出50kHz；若S1开关off，S2开关off，S3开关on的话，不管其他开关是什么，OUTPUT端输出10kHz…，依次类推，如表7-3所示。

表7-3

S1	S2	S3	S4	S5	S6	S7	S8	OUTPUT
0	×	×	×	×	×	×	×	100kHz

续表

S1	S2	S3	S4	S5	S6	S7	S8	OUTPUT
1	0	×	×	×	×	×	×	50kHz
1	1	0	×	×	×	×	×	10kHz
1	1	1	0	×	×	×	×	5kHz
1	1	1	1	0	×	×	×	1kHz
1	1	1	1	1	0	×	×	500 Hz
1	1	1	1	1	1	0	×	100 Hz
1	1	1	1	1	1	1	0	50 Hz

开关on为0，off为1

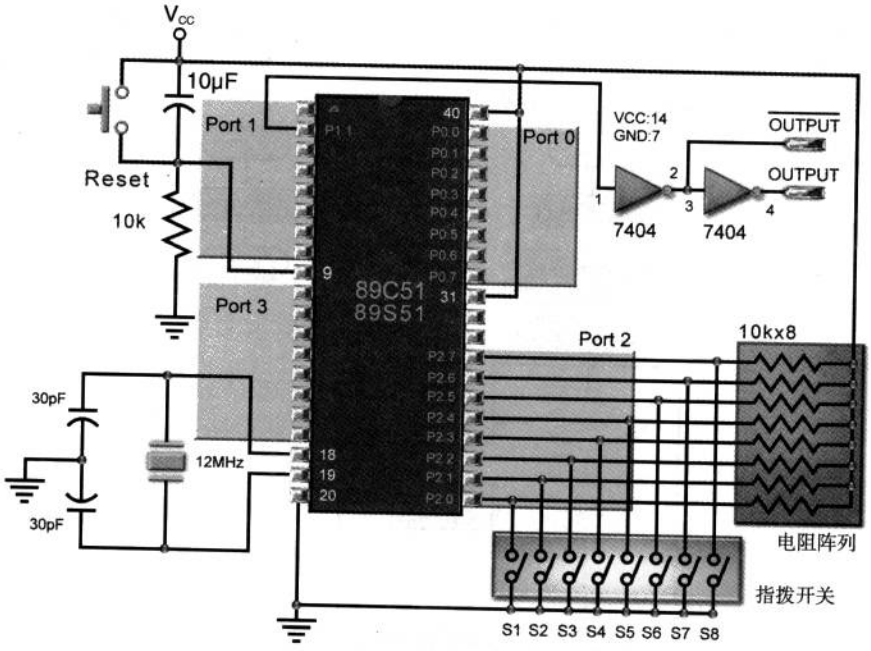


图7-20 简易频率产生器电路图

流程图与程序设计

若f=100kHz，则T=10μs，输出脉冲每5μs就要变化一次，如图所示。

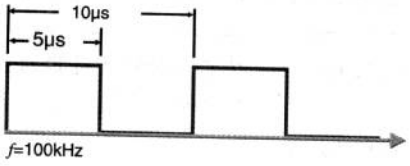


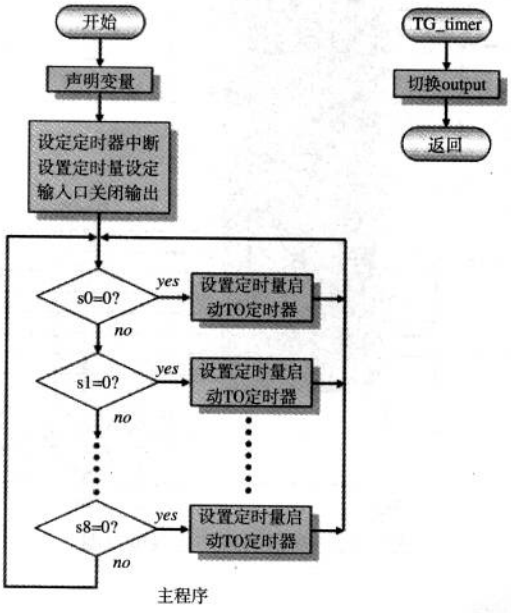
图7-21

同理，对所要产生的信号整理如表7-4所示。

表7-4

频率	周期	变化时间	定时量
100 kHz	10μs	5μs	counts=5
50 kHz	20μs	10μs	counts=10
10 kHz	100μs	50μs	counts=50
5k Hz	200μs	100μs	counts=100
1k Hz	1000μs	500μs	counts=500
500 Hz	2000μs	1000μs	counts=1000
100 Hz	10000μs	5000μs	counts=5000
50 Hz	20000μs	10000μs	counts=10000

因此，在此编写一个定时中断子程序FG_timer，而其定时量将随指拨开关的设定而定时。例如S1接通，则FG_timer的定时量为5，每次5μs中断一次，而中断时，就将输出反相，如此将可得到输出100kHz的频率。



主程序

```
/* ch7_5.c-频率产生器实验，P2的开关控制输出频率 */
//==声明区=====================================
#include        <reg51.h>       // 定义8051寄存器的头文件
sbit output=P1^1;               // 声明输出端
/*声明指拨开关s1~S8 位置*/
sbit s1=P2^0;  sbit s2=P2^1;  sbit s3=P2^2;  sbit s4=P2^3;
sbit s5=P2^4;  sbit s6=P2^5;  sbit s7=P2^6;  sbit s8=P2^7;
```

```
unsigned int  count=0;                          // T0(MODE 1)的计数值
#define          TH_M1(65636-count)/256         // T0(Mode 1)计数高8位
#define          TL_M1(65636-count)%256         // T0(Mode 1)计数低8位
//==主程序==================================
main()                                          // 主程序开始
{    IE=0x82;                                   // 1000 0010，启用TF0中断
     TMOD=0x01;                                 // 0000 0001，T0采用mode1
     output=1;                                  // 输出初值为1
     P2=0xff;                                   // 设计P2输入口
     while(1)                                   // 无穷循环，程序一直运行
     {   if (s1==0)count=5;                     // 按下s1，输出100kHz
         else if(s2==0)count=10;                // 按下s2，输出50kHz
         else if(s3==0)count=50;                // 按下s3，输出10kHz
         else if(s4==0)count=100;               // 按下s4，输出5kHz
         else if(s5==0)count=500;               // 按下s5，输出1kHz
         else if(s6==0)count=1000;              // 按下s6，输出500 Hz
         else if(s7==0)count=5000;              // 按下s7，输出100 Hz
         else if(s8==0)count=10000;             // 按下s8，输出50 Hz
         TH0=TH_M1; TL0=TL_M1;                  // 设置T0计数量高8位、低8位
         TR0=1;                                 // 启动T0
         while(TF0==0);                         // 等待中断(TF0==1)
         TF0=0;                                 // 中断后，清除TF0，关闭T0
         output=~output;                        // 输出反相
     }                                          // while循环结束
}                                               // 主程序结束
```

频率产生器实验（ch7-5.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图7-20连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里所使用的指拨开关，在此将它们的优先等级设成S1>S2>S3>S4，也就是S1开关on的话，其他开关的状态就不考虑；S1开关off，S2开关on的话，其他开关的状态就不考虑……，依次类推，是不是以使用波段开关来代替会比较适当？如何修改电路？

2. 试使用两位数的BCD指拨开关，作为频率的设定，以设计一个能够提供1~99kHz的频率产生器？

7.5.6 计频器实例演练

实验要点

如图7-21所示，所要测试的信号由INPUT输入数字信号，经过7414非门，接到P3.5（即T1引脚）。另外，P3.2连接PB0按钮开关。按一下PB0按钮开关，则进行频率测试。扫描信号与显示信号分别经Port 1、Port 2连接到两位数7段LED数码管模块。

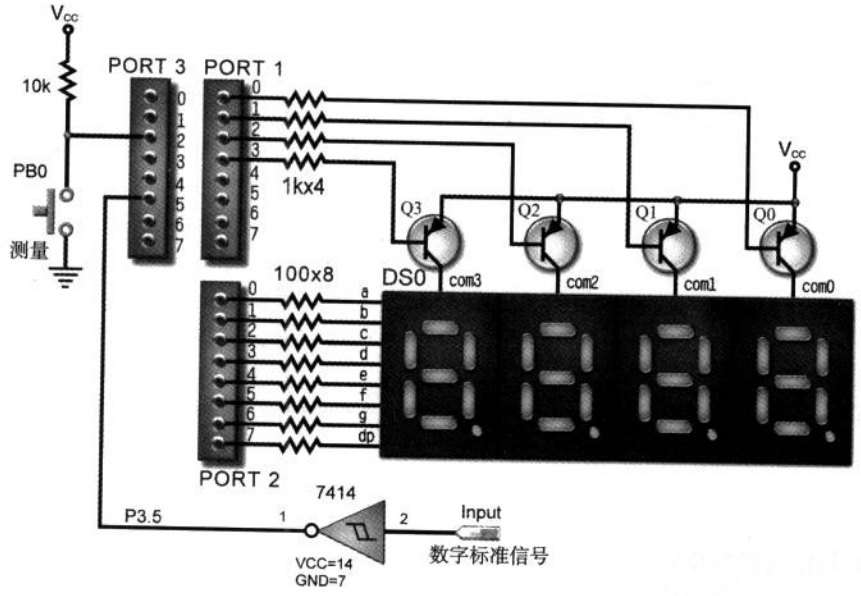


图7-22 简易计频器电路图

流程图与程序设计

基本上分为两阶段，第一阶段执行测量，当PB0按下时，即进入测量阶段。首先关闭显示器，再启动Timer 0定时器定时1s钟，开始定时的同时，Timer 1也开始计数INPUT端的脉冲；而Timer 0定时器定时完成时中断即停止Timer 1的计数，也完成测量阶段。紧接着进入显示阶段，首先将计数量处理一下，放入显示区数组。再将显示区数组送到7段LED数码管模块。直到PB0又被按下，才恢复测量阶段。

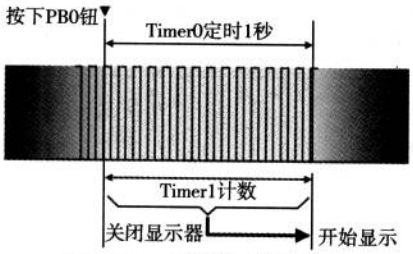
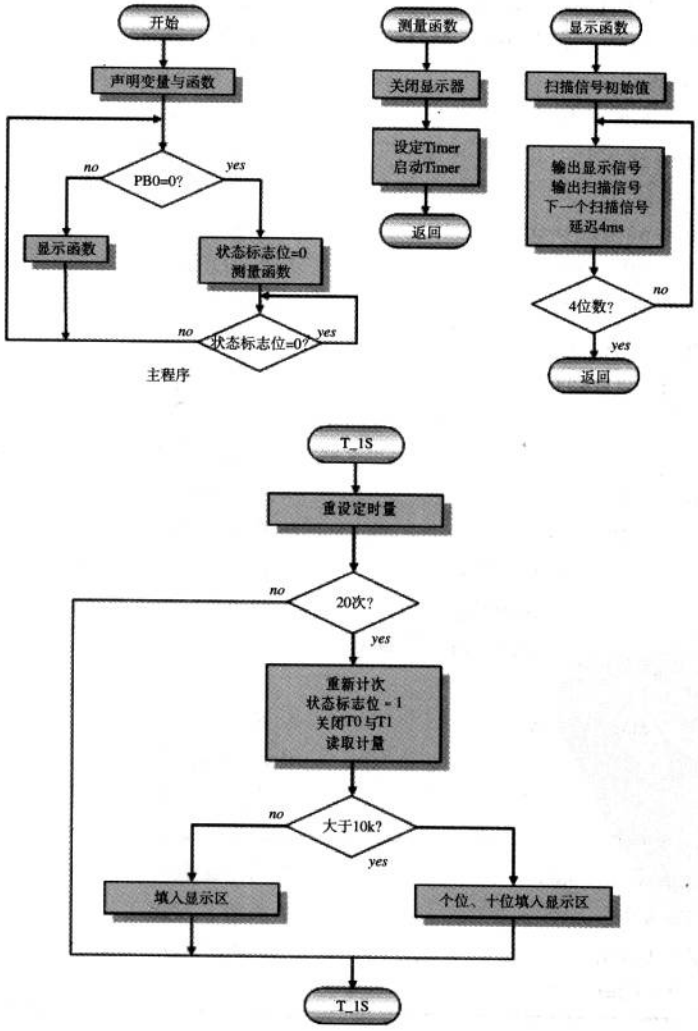


图7-23 计频器动作示意图

由于计数的周期T=1s，所以计数的结果就是赫兹（f=1/THz）。由于只有4位数，范围为0 000~9 999，所以将计数所得的数值区分为10 000以下及10 000~65 535两种，若是10 000以下，则直接显示；若是10 000~65 535，则显示“--nn”，代表nn kHz。流程图与整个程序如下。



```
/* ch7_6.c-计频器实验-由P3.5输入信号，按下PB0于1s后显示其频率 */
//==声明区==================================
#include        <reg51.h>                  // 定义8051寄存器的头文件
#define SEG     P2                         // 定义7段LED数码管接至Port 2
#define SCANP P1                           // 定义扫描线接至Port 1
sbit  PB0=P3^2; // 声明PB0按钮，接至P3.2
char TAB[11]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,     // 数字0-4
               0x92, 0x83, 0xf8, 0x80, 0x98 , 0xbf };     // 数字5-9,及-号
char disp[4]={ 0xc0, 0xc0, 0xc0, 0xc0}; // 声明显示区数组初始显示0000
/*声明T0定时相关声明*/                    //
#define  count_M1   50000                  // T0(MODE 1)的计数值,0.05s
#define  H_0  (65636-count_M1)/256         // T0(MODE 1)计数高8位
#define  L_0  (65636-count_M1)%256         // T0(MODE 1)计数低8位
char times=0;                              // 计算 T0中断次数
/*声明基本变量*/
bit  status_F = 1;                         // 状态标志位
char  scan = 1;                            // 扫描信号
unsigned int freq = 0;                     // 频率变量
sfr16  DPTR = 0x82;                        // 声明DPTR
void delay1m(int);                         // 声明延迟函数
void measure(void);                        // 声明测量函数
void display(void);                        // 声明显示函数
//==主程序==================================
main()                                     // 主程序开始
{    while(1)                              // 无穷循环,程序一直运行
     {   if(PB0==0)                        // 若按下PB0
         {       status_F=0;               // 则进入测量阶段
                 measure();                // 调用测量函数
                 while(status_F==0);       // 等待0，测量完毕
         }
         else display();                   // 显示阶段
     }                                     // while循环结束
}                                          // 主程序结束
//==测量函数================================
void measure(void)                         // 测量函数开始
{    SCANP=0xff;                           // 关闭显示器
     IE=0x8a;                              // 10001010，启用T0、T1中断
     TMOD=0x51;                            //
```

```
    /*0101 0001：T1为计数器、T0为定时器，都采用mode 1*/
    TH0=H_0;TL0=L_0;                    //设置T0计数量的高、低八位
    TH1=0;TL1=0;                        //设置T1归零
    TR0=1;TR1=1;                        //启动T0、T1
}                                       //测量函数结束
//===T0_1S=================================
void T0_1S(void)nterrupt 1              //T0_1S中断子程序开始
    TH0=H_0;TL0=L_0;                    //设置T0计数量的高、低8位
    if(++times==20)                     //若达到1s
    {   times=0;                        //重新计次
        status_F=1;                     //完成测量
        TR1=0;TR0=0;                    //关闭T1、T0
        DPL=TL1;                        //计数量的低8位
        DPH=TH1;                        //计数量的高8位
        freq=DPTR;                      //计数量放入freq变量
        if(freq>=10000)                 //超过10 kHz
        {   disp[3]=TAB[10];            //负号填入千位数显示区
            disp[2]=TAB[10];            //负号填入百位数显示区
            disp[1]=TAB[freq/10000];    //填入十位数显示区
            disp[0]=TAB[（freq/1000）%10];} //填入个位数显示区
        else                                //低于10 kHz
        {   disp[3]=TAB[(freq/1000)%10];    //填入千位数显示区
            disp[2]=TAB[(freq/100)%10];     //填入百位数显示区
            disp[1]=TAB[(freq/10)%10];      //填入十位数显示区
            disp[0]=TAB[freq%10];}          //填入个位数显示区
    }                                       //结束if判断(达到1s)
}                                           //T0_1S中断子程序结束
//===显示函数=============================
void display(void)                      //显示函数开始
{   char i;                             //声明变量
    while(PB0==1)                       //若按下PB0
    {   scan=0x01;                      //初始扫描信号
        for(i=0;i<4;i++)                //扫描4次
        {   SEG=0xff;                   //关闭7段LED数码管
            SCANP=~scan;                //输出扫描信号
            SEG=disp[i];                //输出显示信号
            delay1m(4);                 //延迟4ms
            scan<<=1;                   //下一个扫描信号
```

```
            }                                    // for结束扫描4次
        }                                        // 结束while(按下PB0)
}                                                // 显示函数结束
//===延迟函数=================================
void delay1m(int x)                              // 延迟函数开始
{    int i,j;                                    // 声明变量
     for(i=0;i<x;i++)                            // 连数x次,约xms
          for(j=0;j<120;j++);                    // 数120次,约1ms
}                                                // 延迟函数结束
```

计频器实验（ch7-6.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图7-35连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

思考一下

若要使计频器更精细，应如何处理？

7.5.7 看门狗定时器实例演练

实验要点与程序设计

看门狗定时器和一般的定时器/计数器一样，其运作将独立于其他程序之外。换言之，当系统在执行主程序时，看门狗定时器或一般的定时器/计数器也在背后工作；即使系统的主程序停止响应了，看门狗定时器仍在运作。

当微处理机应用于遥测、监控或通信时，通常执行的程序并不会很复杂，但长时间工作（等待）较容易死机；死机后，又很难以人工使系统复位（比如距离太远或不宜人接近）。这时候，就可利用WDT帮我们监督系统，在系统停止响应时使系统复位。

在本实验里，将利用Timer 0作为复位看门狗定时器的定时器，固定每1ms复位看门狗定时器一次，比看门狗定时器复位系统还早，如此一来，看门狗定时器就没有机会使系统复位了！而主程序可执行我们所要的功能，在此以霹雳灯为例，以验证系统是可用的。

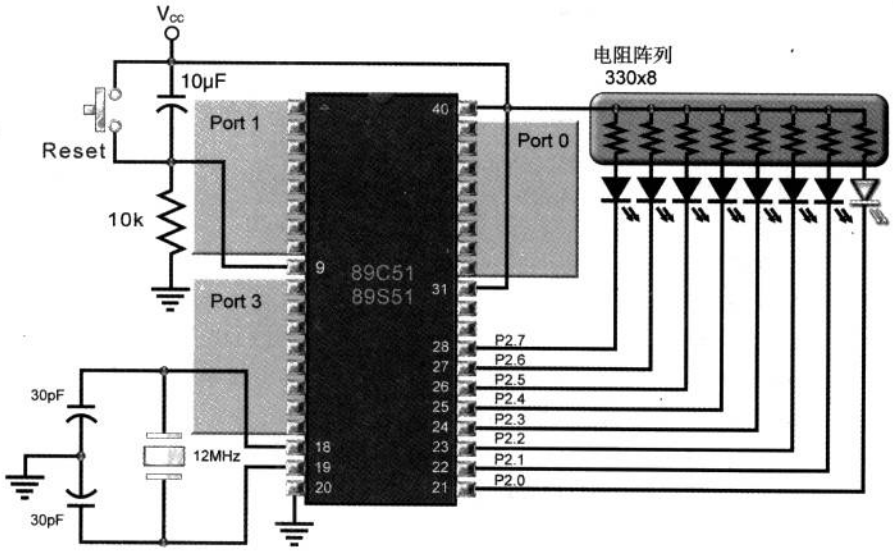
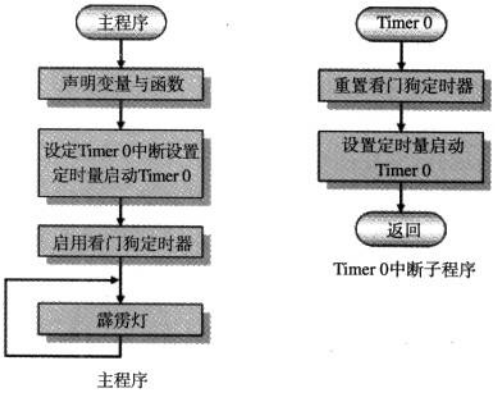


图7-24 霹雳灯实验电路

流程图与程序设计




```
/* ch7_7.c - 看门狗定时器实验 */
//==声明区=====================================
#include      <reg51.h>                    // 定义8051寄存器的头文件
#define LED    P2                          // 定义LED接至Port 2
sfr       WDTRST = 0xa6;                   // 声明WDTRST
int H_0=(65636-1000)/256;                  // 声明T0计数量的高8位
int L_0=(65636-1000)%256;                  // 声明T0计数量的低8位
unsigned char i;                           // 声明无号数字位变量i(占1 B)
void delay(int);                           // 声明延迟函数
```

```
//==主程序=========================================
main()                                          // 主程序开始
{    EA=1; ET0=1; PT0=1;                        // 设定Timer 0中断
     TH0=H_0;                                   // 填入高8位
     TL0=L_0;                                   // 填入低8位
     TR0=1;                                     // 启动Timer 0
     WDTRST = 0x1e;                             // 启用看门狗定时器
     WDTRST = 0xe1;                             // 启用看门狗定时器
     LED=0xfe;                                  // 初值=1111 1110，只有最右1灯亮
     while(1)                                   // 无穷循环，程序一直运行
     {    for(i=0;i<7;i++) // 左移7次
          {    delay(100);                      // 延迟100×5m=0.5s
               LED=(LED<<1)|0x01;               // 左移1位，并设定最低位为1
          }                                     // 左移结束，只有最左1灯亮
          for ( i=0;i<7;i++ )                   // 右移7次
          {    delay ( 100 ) ;                  // 延迟100×5m=0.5s
               LED= ( LED>>1 ) |0x80;           // 右移1位，并设定最高位为1
          }                                     // 结束右移，只有最右1灯亮
     }                                          // while循环结束
}                                               // 主程序结束
//==子程序=========================================
/* 延迟函数，延迟约x×5ms */
void delay(int x)                               // 延迟函数开始
{    int i,j;                                   // 声明整数变量i,j
     for(i=0;i<x;i++)                           // 计数x次,延迟x×5ms
          for(j=1;j<=600;j++);                  // 计数600次，延迟5ms
}                                               // 延迟函数结束
/* Timer 0中断子程序 */
void  feed(void)interrupt 1                     // 中断程序开始
{    TH0=H_0;                                   // 填入高8位
     TL0=L_0;                                   // 填入低8位
     WDTRST = 0x1e;                             // 复位看门狗定时器
     WDTRST = 0xe1;                             // 复位看门狗定时器
}                                               // 中断程序结束
```

看门狗实验（ch7-7.c）

操作

1. 根据上述编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，启动Peripherals/Timer/Timer 0命令，打开Timer/Counter 0窗口；启动Peripherals/Timer/Watchdog命令，打开Watchdog Timer窗口；启动Peripherals/I/O-Ports/Port 2命令，打开Parallel Port 2窗口，如图7-24所示。观察程序执行时，Watchdog Timer窗口里的计数量（Timer字段）是否变化？Timer/Counter 0窗口里的TH0、TL0字段中是否改变？Parallel Port 2窗口里是否有动作？并将它记录在实验报告里。

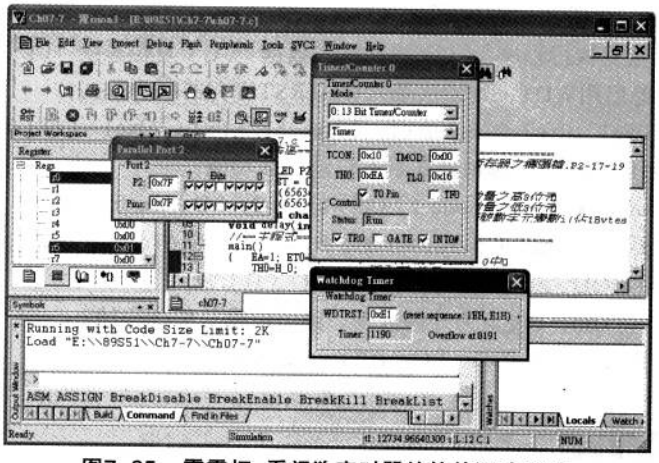


图7-25 霹雳灯-看门狗定时器的软件调试/仿真

3. 若加大Timer 0的计数量，甚至关闭Timer 0，看看是否能正常运作？

思考一下

试想看门狗功能可应用在哪种程序中？

7.6 实时练习



在本章中探讨了8051的定时器/计数器，包括定时器/计数器的结构与设定，定时器/计数器的4种模式，与定时器/计数器相关的寄存器，定时器/计数器程序的编写等，内容非常丰富！在此请试着回答下列问题，以确认对于此部分的掌握程度。



（ ）1. 在8051的Timer里，若使用mode 0，其最大计数量为多少个机器周期？
（A）65 636 （B）8 192 （C）1 024 （D）256

（ ）2. 在12MHz的8051系统里，哪一种模式一次可定时5ms？
（A）mode 0及mode 1 （B）mode 1及mode 2
（C）mode 2及mode 3 （D）mode 3及mode 1

（ ）3. 若要让Timer作为外部计数使用，应如何设定？

（A）Gate=0 （B）Gate=1 （C）C/$\overline{T}$=0 （D）C/$\overline{T}$=1

（ ）4. 如何设定8051的Timer才能从外部引脚启动？

（A）Gate=0 （B）Gate=1 （C）C/$\overline{T}$=0 （D）C/$\overline{T}$=1

（ ）5. 下列何者不是8051所提供的省电模式？

（A）PD模式 （B）IDL模式 （C）LP模式 （D）闲置模式

（ ）6. 89S51的看门狗有何作用？

（A）重复执行程序 （B）找回遗失数据 （C）复位系统 （D）防止中毒

（ ）7. 若要启用WDT，则依次填入WDTRST寄存器哪些数据？

（A）0xe1、0xe2 （B）0xe1、0x1e （C）0x1e、0xe1 （D）0x10、0x01

（ ）8. 8051的Timer的哪种模式具有自动加载功能？

（A）mode 0 （B）mode 1 （C）mode 2 （D）mode 3

（ ）9. 若要设定Timer的模式，可在下列哪个寄存器中设定？

（A）TMOD （B）TCON （C）TH （D）TL

（ ）10. 若将Timer 0设定为外部启动，则可由哪只引脚启动？

（A）P3.2 （B）P3.3 （C）P3.4 （D）P3.5

问答题

1. 试述在12MHz的8051系统里定时器/计数器的四种工作模式，每种模式最多可定时多少时间？

2. 在8051的指令里，若要使用定时器/计数器作为外部计数之用，除了工作模式的选择外，最关键性的设定是什么？

3. 试说明何谓“自动加载”功能？在8051的定时器/计数器里，哪一种工作模式可提供此项功能？

4. 在6.4节的实例演练里，所应用8051定时器/计数器的方式包括“查询方式”及“中断方式”，试说明此两种方式的异同。

5. 若要使用mode 0，试说明设定其定时器/计数器量的方式。

6. 若要使用mode 1，试说明设定其定时器/计数器量的方式。

7. 若要使用mode 2，试说明设定其定时计数量的方式。

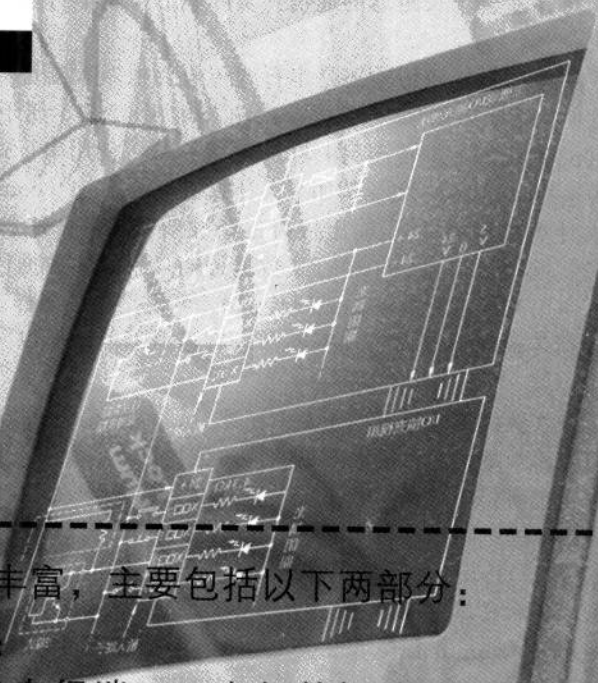
8. 试以图形说明mode 3的工作模式与结构。

9. 在12MHz的8051系统里，若要使用mode 0产生0.5s的延迟，程序应如何编写？

10. 8051系统提供哪两种省电模式？如何进入省电模式？如何唤醒？

11. 试说明89S51的看门狗定时器有何用途？如何启用89S51的看门狗定时器？

第8章 串行口的应用



本章内容丰富，主要包括以下两部分：

硬件部分：

介绍8051串行端口、串行数据转并行数据IC、并行数据转串行数据IC以及MAX232等。

程序与实践部分：

Mode 0串行口的接收与传送、其他工作模式的应用、两台8051小系统的互传应用等。

8.1 数据传输概念



在微型计算机与数字系统里，数据传输可分为并行式传输与串行式传输两种，并行式传输一次传输多个位（通常是8位）。因此，连接两个系统之间的传输线必须有多条。当然一次传输多个位，传输的速度就比较快；但其传输线数比较多，线路费用就要相对提高，线路阻抗匹配、噪声等问题也较多，所以不适合长距离通信。串行式传输每次传输一个位，数据传输的速度表面上不怎么快，但连两个系统之间，只要两条传输线即可，适合于长距离的通信。实际上，目前串行端口传输速率已比并行端口传输速率快。

到底要使用并行式传输还是串行式传输要根据数据量与环境而定。在前面的单元里，不管使用哪个PORT，大都是一次输入/输出8个位，属于近距离的并行式数据传输。若要将8051系统的数据传至另一8051系统，则可使用串行式数据传输。最典型的串行式数据传输接口是RS232C，例如个人计算机里的com1、com2接口就属于RS232C接口。虽然个人计算机里的输入/输出接口逐渐被新一代的USB接口、网卡接口所取代，但USB接口、网卡接口也属于串行式数据传输。串行式数据传输的媒介，除使用铜导线外，还可使用光纤以及电波（无线传输），可说是多种多样，传输速度也日益提高。而并行式数据传输的速度与传输距离却一直迟滞不前，所以并行式数据传输的应用在逐步减少。

在串行式数据传输里有单工及双工之分，单工就是一条线只能有一种用途，例如输出线就只能将数据传出，输入线就只能将数据传入。而双工就是一条线有两种用途，可传入数据，也可传出数据。若系统上只有一条传输线，且在同一时刻中不是进行传入数据就是传出数据，则称之为“半双工”。若在系统上有两条传输线，这两条传输线可同时进行数据传入与传出，则称之为“全双工”。

通常以每秒传输多少位（**bit per second**，简称**bit/s**）表示串行式数据传输的速率，又称为比特率（baud rate）。8051提供一个全双工的万用异步串行端口（Universal Asynchronous Receiver-Transmitter，简称UART），这个串行端口有4种工作模式（mode），使用不同工作模式其比特率各有不同，说明如下。

- **mode 0**：由8051主导，其比特率为8051系统时钟脉冲的1/12，即OSC/12，以12MHz的系统为例，则其比特率为1M bit/s。
- **mode 1**或**mode 3**：此模式为可变比特率的串行口，主要是为了配合所连接系统的时序，以达到不同系统的数据传输。
- **mode 2**：此模式提供两种不同比特率的选择，即OSC/32或OSC/64，其中的OSC为8051系统时钟脉冲。

通常，微控制器里的数据处理属于并行式处理。对8051而言，一次处理一个字节，也就是8个位，不管怎样，串行式数据与并行式数据之间的转换是无法避免的。在8051里，若要把8位的并行数据传出去，只要把数据放入串行寄存器（SBUF）即可，8051就会帮我们把这些数据一个位一个位送出去。接收串行数据也是一样，8051会把外面传入的数据，一个位一个位放入SBUF，当SBUF存满后，即为并行数据。

8.2 认识74164/74165



若从8051的串行端口将数据传出后，则可以通过串行转并行的IC（如74 164）将串行数据转换成并行数据；若要把并行数据由8051的串行口传入8051，则需要经并行转串行的IC（如74 165）将并行数据转换成串行数据，即可传入8051。下面分别对74 164和74 165进行介绍。

74164

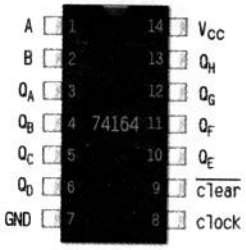


图8-1 74164引脚图

如图8-1所示，74164是一片能够进行串行转并行的IC，对其中各引脚说明如下。

- $\overline{clear}$：本引脚为清除引脚，当此引脚为低电平时，则并行输出引脚将全部变为低电平。
- **A**、**B**：串行数据输入引脚，这两只引脚完全一样，我们可将这两只引脚连接在一起，再接到串行数据源。也可将其中一脚连接到V_{CC}，另一只引脚连接串行数据源。
- **clock**：本引脚为时钟脉冲引脚，而此IC为正边缘触发型，也就是输出引脚的状态变化是发生在时钟脉冲由低电平变为高电平的时候。74164时序图如图8-2所示。

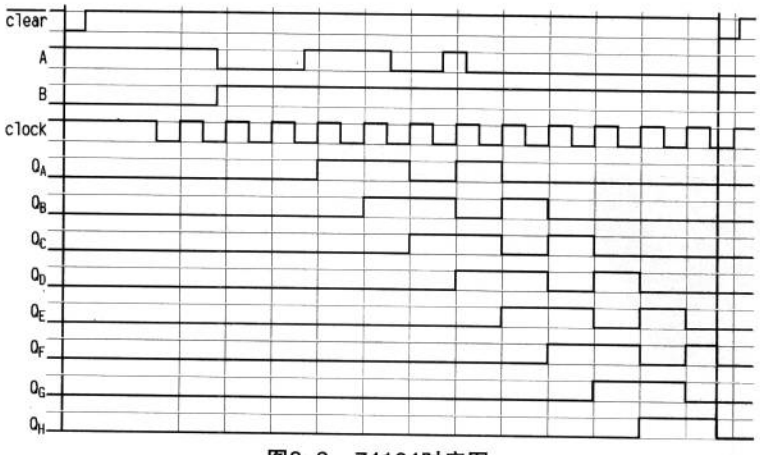


图8-2 74164时序图

- **QA~QH**：数据输出引脚，这8只引脚为b_0~b_7的8位并行数据输出引脚。

74165

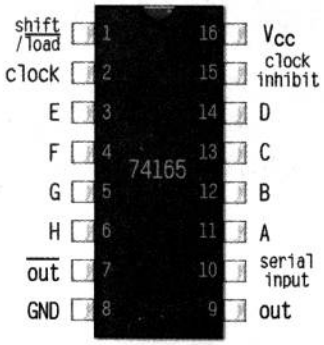


图8-3 74165引脚图

如图8-3所示，74165是一片能够进行并行转串行的IC，对其中各引脚说明如下。

- shift/$\overline{\text{load}}$：本引脚为数据加载与移位控制引脚，当此引脚为低电平时，则并行输入引脚的状态将全部被加载。当此引脚为高电平时，即可随时钟脉冲进行移位式串行输出。
- **clock inhibit**：本引脚为时钟脉冲禁止引脚，当本引脚为高电平时，输出引脚将不随时钟脉冲而变化。当本引脚为低电平时，输出引脚将随时钟脉冲进行变化移位式串行输出。
- **clock**：本引脚为时钟脉冲引脚，而此IC为正缘触发型，也就是输出引脚的状态变化是发生在时钟脉冲由低电平变为高电平的时候。
- **serial input**：本引脚为串行输入引脚，若要进行并行数据输入到串行数据输出的转换，则保持此引脚为低电平即可。若要进行串行数据输入到串行数据输出的转换，则此引脚连接串行数据源。
- **A~H**：并行数据输入引脚，这8只引脚为b_0~b_7的8位并行数据输入引脚，接到并行数据源。
- **out**：本引脚为串行数据输出引脚。
- $\overline{\text{out}}$：本引脚为串行数据反相输出引脚。

74165的真值表如表8-1所示。

表8-1

输入					输出	
shift / $\overline{\text{load}}$	clock inhibit	clock	serial input	并行输入 A~H	out	$\overline{\text{out}}$
0	×	×	×	a~h	h	$\overline{\text{h}}$
1	0	0	×	×	Q_{ho}	$\overline{Q_{ho}}$
1	0	↑	1	×	Q_n	$\overline{Q_n}$
1	0	↑	0	×	Q_n	$\overline{Q_n}$
1	1	↑	×	×	Q_{ho}	$\overline{Q_{ho}}$

8.3 认识MCS-51的串行口



MCS-51的串行口为全双工的串行口，可接收并传送串行数据，不管是接收还是传送，都使用串行寄存器（SBUF）。接收串行数据时，所输入的串行数据将存入SBUF，存满之后产生中断，再将SBUF里的8位数据移做他用。传送数据时，先将所要送出的数据放入SBUF，即可一个位一个位地传到目的地。很明显，SBUF扮演了关键性的角色，在8051中，接收用的SBUF与传送用的SBUF虽然都叫做SBUF，但是它们分别是两个不同地址的8位寄存器。不过，它对于使用者的操作没有影响。

MCS-51的串行端口有以下4种工作模式。

mode 0

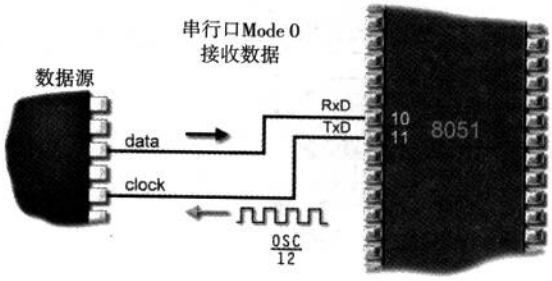


图8-4 接收数据

mode 0工作模式是以固定比特率的移位式数据传输，其比特率为OSC/12，若在时钟脉冲为12MHz，则其比特率为1Mbit/s。在此模式下，不管是接收数据还是传送数据，CPU的RxD引脚（P3.0）连接串行数据线，TxD引脚（P3.1）连接移位脉冲线。执行数据接收时，由TxD引脚送出移位脉冲，而由RxD引脚收下串行数据，如图8-4所示；执行数据传送时，也是依据TxD引脚所送出的移位脉冲，而由RxD引脚送送数据串行数据，如图8-5所示。

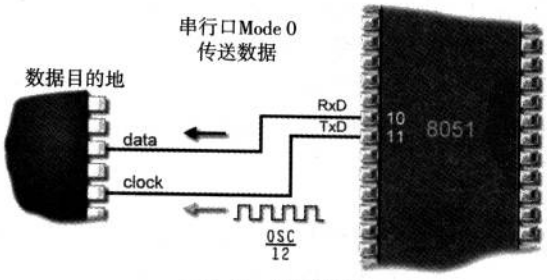


图8-5 传送数据

如图8-6所示为8051内部串行口的结构，只要将所要送出的数据放入SBUF，例如“SBUF=0x12”，则0x12写入SBUF的动作会在S6的2周期发生，将“1”放入SBUF左边的触发器（看作第9位），以作为结束传送的标识位，同时启动TX控制器的开始信号，而TX

控制器的送出信号输出1，则TxD引脚即可输出移位脉冲，开始传出数据。紧接着，SBUF内的数据依次移位经RxD引脚输出，同时在SBUF左边补入0。当送出8个位时，SBUF里除最右边为1外，其余皆为0，零检测电路（只检测SBUF左边7个位）将检测出0的信号，以驱动TX控制器，TX控制器随即产生一个TI中断信号，并将送出信号输出为0，停止传送。

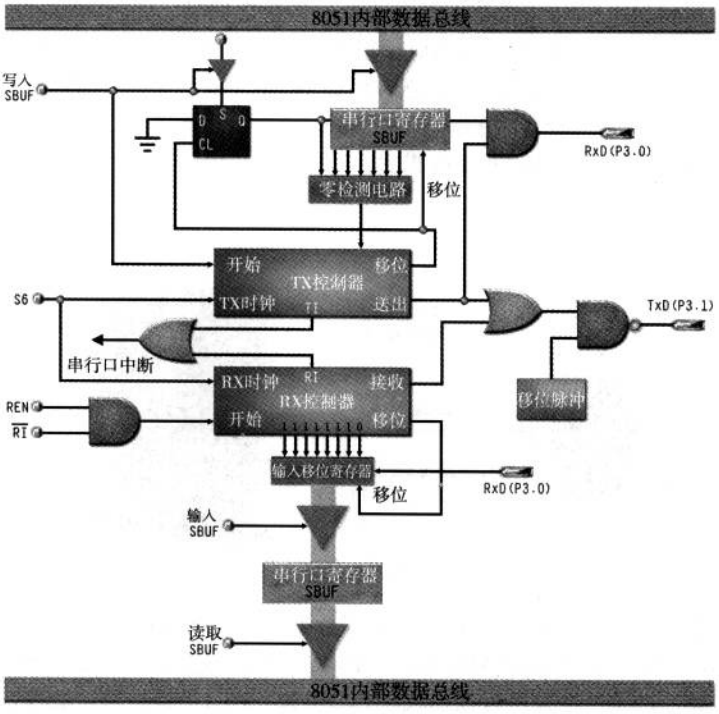


图8-6 串行口结构

若要进行数据接收，则需要先将SCON寄存器的REN位设定为1，RI位设定为0，才能启动RX控制器的开始信号，RX控制器将“11111110”加载输入移位寄存器。紧接着，RX控制器的接收信号输出1，则TxD引脚即可输出移位脉冲，开始接收数据。每当一个位串行数据经RxD引脚输入到输入移位寄存器时，RX控制器将送出一个移位信号，让输入移位寄存器左移一位。当输入移位寄存器最左位为0时，RX控制器将再进行一个位的输入，然后产生一个RI中断信号，并将接收信号输出为0，停止接收。

mode 1

mode 1工作模式是以可变的比特率进行串行数据的传输，其比特率可由Timer 1来控制（若是8052则还可使用Timer 2控制比特率）。在此模式下，8051的RxD引脚连接数据源的TxD引脚，8051的TxD引脚连接目的地的RxD引脚。

在mode 1下，每个数据是由10位组成，包括起始位（start bit）、8个位的数据以及停止位（stop bit），其中第一个位就是低电平的起始位，紧接着是8位数据的bit 0（即LSB），bit 7（MSB）之后是高电平的停止位，如图8-8所示。

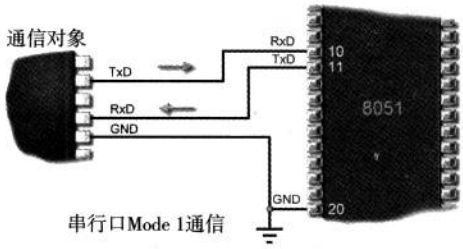


图8-7 mode 1串行数据传输



图8-8 mode 1的数据串格式

mode 2

mode 2工作模式是以OSC/32或OSC/64的比特率进行串行数据的传输，而其线路的连接，也是8051的RxD引脚连接数据源的TxD引脚，8051的TxD引脚连接目的地的RxD引脚。在mode 2下，每笔数据是由11位组成，包括起始位（start bit）、8个位的数据、奇偶位（parity bit）以及停止位（stop bit），其中第一个位就是低电平的起始位，紧接着是8位数据的bit 0（即LSB），而bit 7之后的是奇偶位，最后则是高电平的停止位，如图8-9所示。

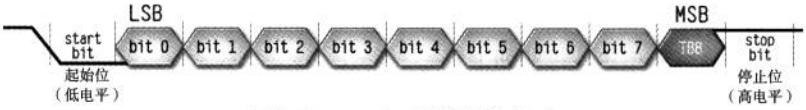


图8-9 mode 2的数据串格式

当进行数据传出时，第9个位TB8（即SCON寄存器中的TB8）为奇偶位，可取自程序状态字组寄存器PSW中的P位，以达到奇偶检验的目的。当接收数据时，第9个位将直接移入SCON寄存器中的RB8，而不必管停止位。

mode 3

mode 3工作模式是以可变的比特率进行串行数据的传输，其比特率可由Timer 1来控制（若是8052则还可使用Timer 2控制比特率）。除此之外，mode 3与mode 2几乎完全一样。

8.4 认识SCON串行口控制寄存器

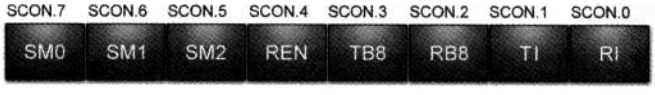


图8-10 SCON寄存器

串行端口控制寄存器（serial port control register，简称**SCON**）是一个8位、可位寻

址的寄存器，如图8-10所示其功能是设定与控制串行端口。对其中的各位说明如下。

SM0与SM1

这两个位的功能是设定串行端口的模式，如表8-2所示。

表8-2

SM1	SM0	mode	功能简介	比特率
0	0	0	移位寄存器	OSC/12
0	1	1	8位UART	可变
1	0	2	9位UART	OSC/32或OSC/64
1	1	3	9位UART	可变

SM2

本位为多重处理器通信启用位，说明如下：

- mode 0时，SM2=0；
- mode 1时，若SM2=1，且收到有效的停止位，则RI=1（产生RI中断），否则RI=0；
- mode 2或mode 3时，若SM2=1，且收到的第9位为1，则RI=1（产生RI中断），若第9位为1，则RI=0。

REN

本位为串行接收启用位，说明如下：

- REN=1，开始接收；
- REN=0，停止接收。

TB8

mode 2或mode 3传送数据时，本位为第9传送位，可用软件来设定或清除。

RB8

- mode 2或mode 3接收数据时，本位为第9个接收位；
- mode 1时，若SM2=0，则本位为停止位；
- mode 0时，本位无作用。

TI

本位为传送中断标志位，当中断结束时，本位并不会恢复为0，必须由软件清除。

- mode 1、mode 2或mode 3时，若完成传送停止位，则本位自动设定为1，并产生TI中断。
- mode 0时，若完成传送第8位，则本位自动设定为1，并产生TI中断。

RI

本位为接收中断标志位，当中断结束时，本位并不会恢复为0，必须由软件清除。

- mode 1、mode 2或mode 3时，若完成接收到停止位，则本位自动设定为1，并产生RI中断。
- mode 0时，若完成接收第8位，则本位自动设定为1，并产生RI中断。

8.5 比特率设定



8051串行口的比特率设定方式有如下几种。

mode 0

在mode 0下，比特率固定为OSC/12，不需要进行设定！而是完全依照系统的时钟脉冲而定，不是软件所能改变的。

mode 2

在mode 2下，其比特率可为OSC/32或OSC/64，即：

$$\text{比特率} = \frac{2^{SMOD}}{64} \times SOC$$

其中SMOD为PCON寄存器（详见7.3.3节）中的bit 7，若将SMOD设定为0，则设定采用的比特率OSC/64；若将SMOD设定为1，则设定采用的比特率OSC/32。以12MHz的系统为例，

$$\text{若SMOD=0，则波特率} = \frac{2^0}{64} \times 12M = 187.5k\ (\text{bit/s})$$

$$\text{若SMOD=1，则波特率} = \frac{2^1}{64} \times 12M = 375k\ (\text{bit/s})$$

mode 1或mode 3

在mode 1或mode 3下，比特率可由Timer 1（8052则还可选择Timer 2）的溢出脉冲控制，以Timer 1采用具自动加载功能的mode 2为例，其产生的比特率为：

$$\text{比特率} = \frac{2^{SMOD}}{32} \times \frac{OSC}{12 \times (256 - TH1)}$$

在11.0592MHz的系统下，若想要产生19.2kbit/s的比特率，且SMOD=1，则

$$19\ 200 = \frac{2^1}{32} \times \frac{11\ 059\ 200}{12 \times (256 - TH1)} = \frac{11\ 059\ 200}{16 \times 12 \times (256 - TH1)}$$

256-TH1=3，TH1=253=0xfd

表8-3所示的是使用Timer 1时常用的比特率设定。

表8-3

比特率	OSC	SMOD	TH1
62.5k	12MHz	1	0xff
19.2k	11.0592MHz	1	0xfd
9600	11.0592MHz	0	0xfd
4800	11.0592MHz	0	0xfa
2400	11.0592MHz	0	0xf4
1200	11.0592MHz	0	0xe8
137.5	11.0592MHz	0	0x1d

8.6 认识MAX232



如果是短距离的串行数据传输，则标准的TTL或CMOS就足以应付；若要进行长距离的串行数据传输，使用标准的TTL或CMOS恐怕驱动能力不足，且噪声容限（Noise Margin）太小，通信质量很差！RS-232是一种可长距离传输的通信方式，因此，必须突破噪声容限太小与驱动能力不足的限制，于是出现了相关的驱动IC，Maxim公司的MAX232系列就属这类IC，以其MAX232为例，如图8-11所示，这片IC提供RS232传送与接收的驱动，在传送方面，MAX232内部将+5V电源提升为+10V及-10V，然后接收TTL/CMOS的+5V电平，转换成?10V的信号，再送到线路上。在接收方面，MAX232从线路上接收+10V的信号，经内部寄存器，转换成TTL/CMOS的+5V电平。简单地说，MAX232只不过是个电平转换装置而已，但只要+5V电源与4~5个小电容（10μF即可）就能同时提供双向的电平调整。对于使用者而言，把它当成一般的寄存器来使用即可。RS232线路如图8-12所示。

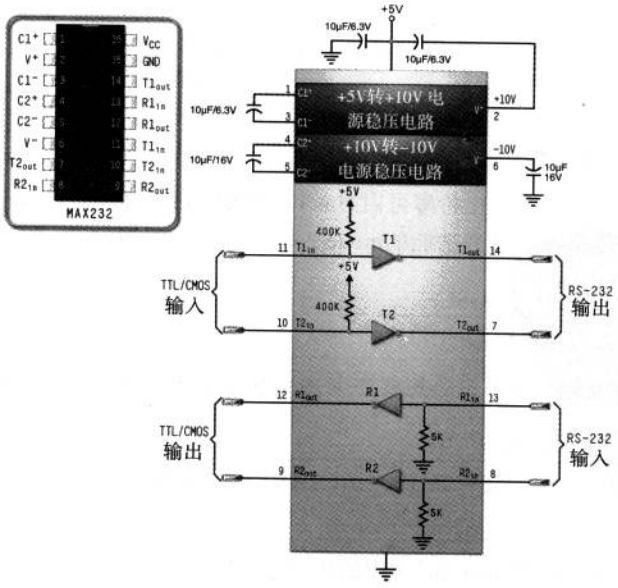


图8-11 MAX232

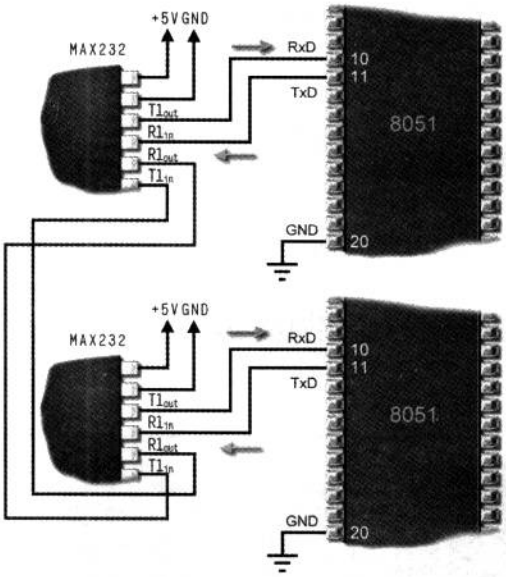


图8-12 RS232线路

8.7 多处理器间的数据传输



前面所介绍的串行式数据传输是一对一的通信，也就是说A系统通过两条传输线与B系统通信。而8051也支持多处理器之间的数据传输，如下图所示，主处理器（master processor）与数个从处理器（slave processor）构成一个主从式结构，每个处理器的TxD线都接在一起，而RxD线也接在一起，对于每个从处理器都给予编码寻址。

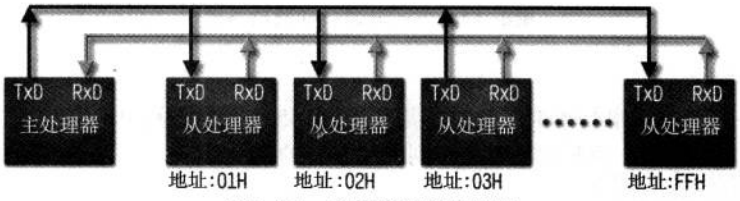


图8-13 主从式处理器结构

当8051的串行口工作于mode 2或mode 3时，若主处理器要传一笔数据给某个从处理器时，先送出目的从处理器的地址字节（address byte），而数据字节与地址字节的差别在于其第9位，若第9位为1，代表该笔数据为地址；若第9位为0，代表该笔数据为数据。若从处理器的SM2位为1时，如接收到的是地址字节，将会产生中断；如接收到的是数据字节，则不会产生中断。因此，当主处理器送出地址字节时，所有从处理器都将产生中断，然后各自判断所接收到的地址，是否就是自己的地址？若是（表示接下来的数据就是要给它的），则将SM2位设为0，然后准备接收下一笔数据。而其他非此地址的从处理器，

将不接收下一笔数据。关于多处理器的通信，限于篇幅，在此无法详细说明，请参阅串行口通信的相关书籍。

8.8 实例演练



在本单元里提供6个范例，如下所示。

8.8.1 移位式数据串入实例演练

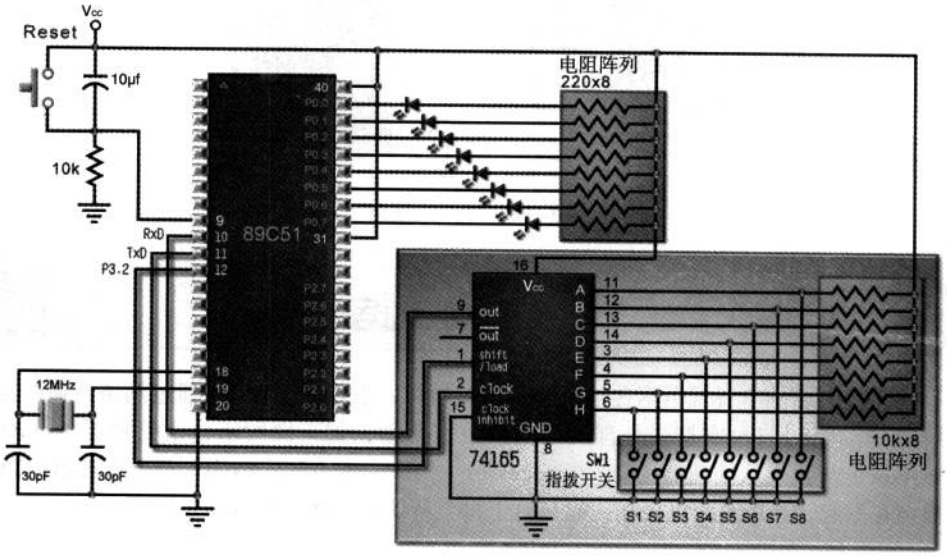


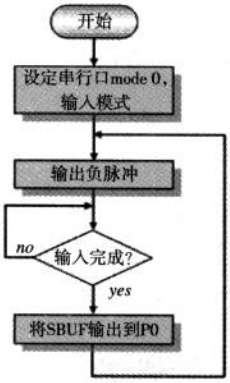
图8-14 并行转串行实验电路图

实验要点

如图8-14所示，串行数据的数据源是利用74165将指拨开关DIP SW的状态转换成串行数据而由RxD输入，而其状态将反映到P0所连接的LED上。

流程图与程序设计

根据74165的真值表，若要将并行数据加载到74165，则需要给74165的第1脚（shift/ $\overline{\text{load}}$ ）一个负脉冲，在此利用P3.2引脚连接到74165的第1脚，利用“P3^2=0;”将P3.2引脚变为低电平，再用两个“if（p3^2==0）{};”多延迟一点时间，最后再用“P3^2=1;”将P3.2引脚恢复为高电平。74165加载DIP SW上的并行数据后，随即依照8051通过TxD引脚传来的移位脉冲，将数据一位一位地由RxD引脚传入8051。当SBUF满了，即产生RI中断，我们只要将SBUF寄存器里的数据复制到P0即可。




```
/* ch8_1.c – Mode 0串行输入实验 */
#include <reg51.h>                          // 包含reg51.h文件
sbit    load=P3^2;                          // 声明P3^2位置
main()                                      // 主程序开始
{ SCON=0x11;                        // 设定为mode 0，REN=1，RI=1
//====b7===b6===b5===b4===b3===b2===b1===b0===
//===SM0==SM1==SM2==REN==TB8==RB8===TI===RI===
//=====0====0====0====1====0====0====0====1===
  while(1)                                  // while循环开始
  {       load=0;                           // 输出负脉冲，让74165加载数据
          if(load==0);                      // 延长负脉冲宽度
          if(load==0);                      // 延长负脉冲宽度
          load=1;                           // 恢复高电平
          RI=0;                             // 清除RI
          while(RI==0);                     // 等待RI串行输入中断
          P0=SBUF;                          // RI=1时(接收完成)，输出至P0
  }                                         // while循环结束
}                                           // 主程序结束
```

Mode 0串行输入实验（ch8–1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图8–14连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

思考一下

1. 在本实验里，有没有“抖动”的困扰？

2. 若希望指拨开关中的S1、S3、S53个开关都on，则前4个LED亮；S2或S4或S6开关on，则后4个LED亮；S7及S8开关on，则所有LED全亮，程序应如何编写？

8.8.2 移位式数据串出实例演练

实验要点

如图8-15所示，串行数据由8051的RxD引脚连接到74164的A、B输入引脚，而8051的TxD引脚连接到74164的clock引脚，形成一个串行输出的电路。当然，74164的 $\overline{\text{clear}}$ 引脚必须连接到V_{CC}，74164才会正常工作。在此我们希望将8P指拨开关DIP SW的状态，经Port 2输入到8051，再由8051通过上述串行路径，输出到74164，而74164的并行输出引脚连接8个LED，让这8个LED反应指拨开关的状态。

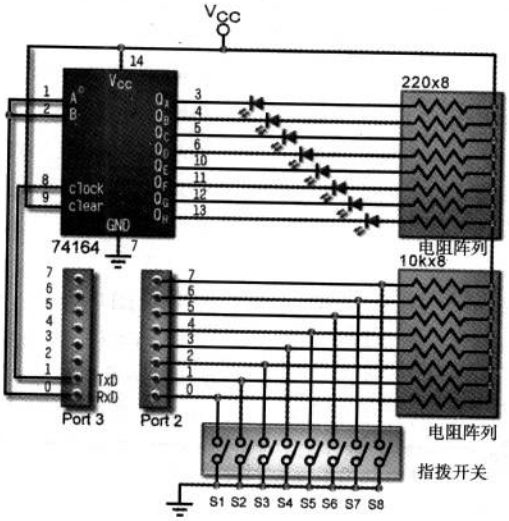
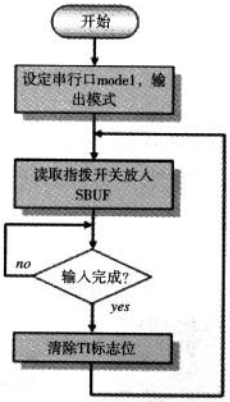


图8-15 串行转并行实验电路图

流程图与程序设计

在此要以mode 0进行数据的串行输出，所以将SCON设定为0，即为mode 0。紧接着从P2读取指拨开关的状态，然后把它放入SBUF，CPU即进行串行数据的输出。程序只要等待TI中断，表示SBUF里的数据以全部传出，再重新读取Port 2的开关状态，进行同样的串行输出即可。




```
/* ch8_2.c – Mode 0串行输出实验 */
#include  <reg51.h>                          // 包含reg51.h文件
main()                                       // 主程序开始
{ SCON=0;                                    // 设定为mode 0
//====b7===b6===b5===b4===b3===b2===b1===b0===
//===SM0==SM1==SM2==REN==TB8==RB8===TI===RI===
//=====0====0====0====0====0====0====0====0===
  P2=0xff;                                   // 规划Port 2为输入口
  while(1)                                   // while循环开始
  {       SBUF=P2;                           // 将指拨开关状态，放入SBUF
          while(TI==0);                      // 等待TI串行输出中断
          TI=0;                              // TI=1时(传送完成)，清除TI
  }                                          // while循环结束
}                                            // 主程序结束
```

Mode 0串行输出实验（ch8-2.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图8-15连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

8.8.3 mode 1 实例演练

实验要点

如图8-16所示，我们将89C51的第10脚与第11脚短路，让串行数据输出连接到串行输入，以达到自己传给自己的目的。另外，第11脚所要传出的数据是来自Port 2所连接的指拨开关DIP SW状态；而第11脚所接收到的串行数据将反应到P0所连接的LED上。

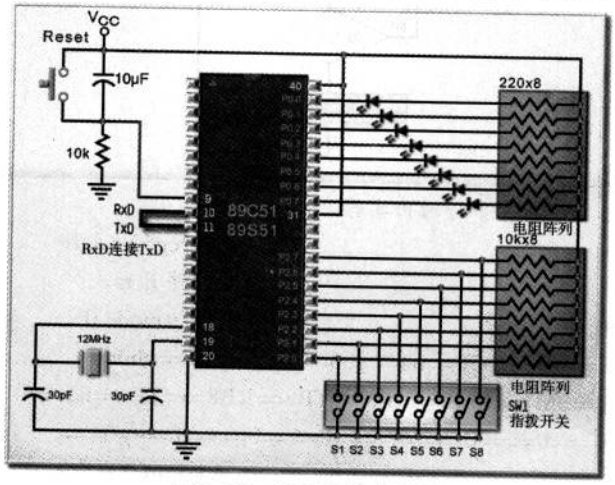
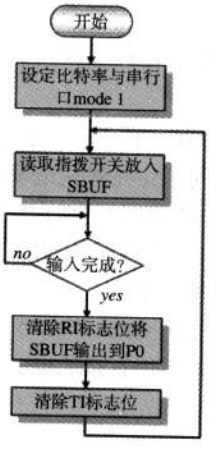


图8-16 自传实验电路图

流程图与程序设计

在此所要采用的比特率为9 600bit/s。首先利用TMOD寄存器将Timer 1设定为Mode 2，再将PCON寄存器的SMOD位设定为0，然后TH1寄存器加载0xfd（即253），最后启动Timer 1，即可产生9 600bit/s的比特率。另外，在SCON寄存器里，将串行端口设定为mode 1。这样，由Port 2所读取的数据，放入SBUF寄存器（传送至SBUF寄存器），CPU即自动传送。另一方面，CPU也自动接收，当接收的SBUF寄存器满了，即产生RI中断，我们只要将SBUF寄存器里的数据复制到P0即可。



```
/* ch8_3.c – Mode 1实验 */
#include <reg51.h>                          // 包含reg51.h文件
sbit SMOD = 0x87^7;                         // SMOD为PCON的bit7
main()                                      // 主程序开始
{ P2=0xff;                                  // 规划Port 2为输入口
  TMOD=0x20;
  /*将Timer 1设定mode 2，作为比特率产生*/
  SMOD=0;                                   // 将SMOD设定为0
  TH1=0xfd;                                 // 比特率设定为9600bit/s
  TR1=1;                                    // 启动Timer 1
 SCON=0x50;                                 // 设定为mode 0
//====b7===b6===b5===b4===b3===b2===b1===b0===
//===SM0==SM1==SM2==REN==TB8==RB8===TI===RI===
//=====0====1====0====1====0====0====0====0===
 while(1)                                   // while循环开始
 {      SBUF=P2;                            // 将指拨开关状态，放入SBUF
        while (RI==0);                      // 检查是否完成接收?
        RI=0;                               // RI=1时(接收完成)，清除RI标志位
        P0=SBUF;                            // 将所接收的数据输出到Port 0
        TI=0;                               // 清除TI标志位
 }                                          // while循环结束
}                                           // 主程序结束
```

Mode 1实验（ch8–3.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图8–16连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

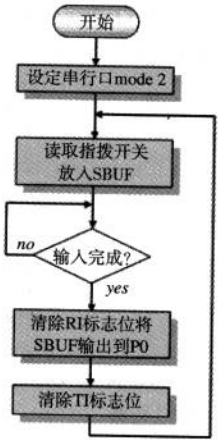
8.8.4 mode 2 实例演练

实验要点

如图8-16所示，其功能与8.8.3节相同，只是在此采用mode 2工作模式。

流程图与程序设计

在比特率的设定方面，串行口的mode 2比mode 1简单！只有OSC/64及OSC/32两种，若SMOD位设定为0，则为OSC/64、SMOD位设定为1，则为OSC/32。在此采用较快的比特率（OSC/32），因此，将PCON寄存器的SMOD位设定为1，然后在SCON寄存器里，将串行端口设定为mode 2，REN为1。即可由Port 2所读取的数据，放入SBUF寄存器（传送至SBUF寄存器），CPU即自动传送。另一方面，CPU也自动接收，当接收的SBUF寄存器满了，即产生RI中断，我们只要将SBUF寄存器里的数据复制到P0即可。




```
/* ch8_4.c – Mode 2实验 */
#include <reg51.h>                          // 包含reg51.h文件
sbit SMOD = 0x87^7;                         // SMOD为PCON的bit7
main()                                      // 主程序开始
{ P2=0xff;                                  // 规划Port 2为输入口
  SMOD=1;                                   // 将SMOD设定为1
  SCON=0x90;                                // 设定为mode 2
//====b7===b6===b5===b4===b3===b2===b1===b0===
//===SM0==SM1==SM2==REN==TB8==RB8===TI===RI===
//=====1====0====0====1====0====0====0====0===
 while(1)                                   // while循环开始
 {      SBUF=P2;                            // 将指拨开关状态，放入SBUF
        while(RI==0);                       // 检查是否完成接收?
```

```
            RI=0;                    // RI=1时(接收完成)，清除RI标志位
            P0=SBUF;                 // 将所接收的数据输出到Port 0
            TI=0;                    // 清除TI标志位
        }                            // while循环结束
    }                                // 主程序结束
```

Mode 2实验（ch8-4.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图8-16连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

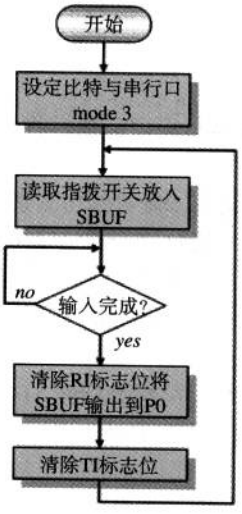
5. 编写实验报告。

8.8.5 mode 3 实例演练

实验要点

如图8-16所示，其功能与8.8.3节相同，只是在此采用mode 3工作模式。

流程图与程序设计



在此所要采用的比特率也是9600bit/s，因此，除了将mode 2的设定改成mode 3外，本程序与ch8-4.c大致相同。

```
/* ch8_5.c - Mode 3实验 */
#include <reg51.h>                          // 包含reg51.h文件
sbit SMOD = 0x87^7;                         // SMOD为PCON的bit7
main()                                      // 主程序开始
{ P2=0xff;                                  // 规划Port 2为输入口
  TMOD=0x20;
  /*将Timer 1设定mode 2，作为比特率产生*/
  SMOD=0;                                   // 将SMOD设定为0
  TH1=0xfd;                                 // 比特率设定为9600bit/s
  TR1=1;                                    // 启动Timer 1
  SCON=0xd0;                                // 设定为mode 3
//====b7===b6===b5===b4===b3===b2===b1===b0===
//===SM0==SM1==SM2==REN==TB8==RB8===TI===RI===
//=====1====1====0====1====0====0====0====0===
  while(1)                                  // while循环开始
  {       SBUF=P2;                          // 将指拨开关状态，放入SBUF
          while(RI==0);                     // 检查是否完成接收?
          RI=0;                             // RI=1时(接收完成)，清除RI标志位
          P0=SBUF;                          // 将所接收的数据输出到Port 0
          TI=0;                             // 清除TI标志位
  }                                         // while循环结束
}                                           // 主程序结束
```

Mode 3实验（ch8-5.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图8-16连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

8.8.6 8051 vs 8051实例演练

实验要点

如图8-17所示，“8051系统一”与“8051系统二”为两个独立的小系统，通过TxD与RxD传输线进行全双工的数据传输，将8051系统一的指拨开关状态，经串行端口传输到8051系统二P0上的LED。同时，将8051系统二的指拨开关状态，经串行端口传输到8051系统一P0上的LED。

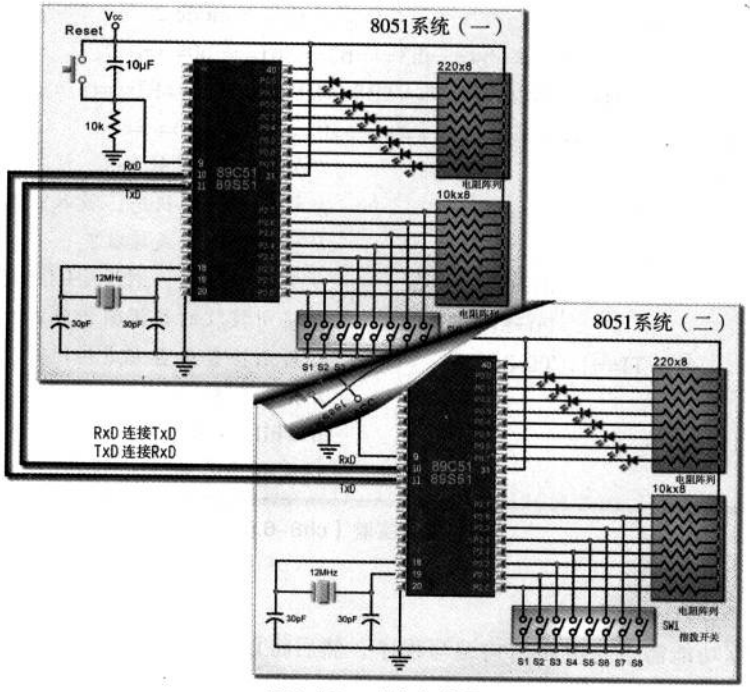
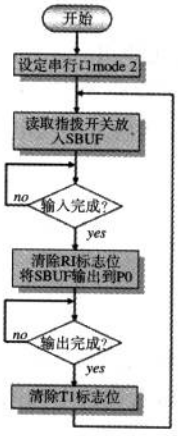


图8-17 对传电路图

流程图与程序设计

在两个89C51里都必须执行下列程序：



```
/* ch8_6.c – 8051互传实验 */
#include <reg51.h>                          // 包含reg51.h文件
sbit SMOD = 0x87^7;                         // SMOD为PCON的bit7
main()                                      // 主程序开始
{ P2=0xff;                                  // 规划Port 2为输入口
  SMOD=1;                                   // 将SMOD设定为1
  SCON=0x90;                                // 设定为mode 2
//====b7===b6===b5===b4===b3===b2===b1===b0===
//===SM0==SM1==SM2==REN==TB8==RB8===TI===RI===
//=====1====0====0====1====0====0====0====0===
  while(1)                                  // while循环开始
  {       SBUF=P2;                          // 将指拨开关状态，放入SBUF
          if(RI==1)                         // 检查是否完成接收?
          {         RI=0;                   // 若完成接收，则清除RI标志位
                    P0=SBUF;}               // 将所接收的数据输出到Port 0
          if(TI==1) TI=0;                   // 否则检查是否完成传送?
                                            // 清除TI标志位
  }                                         // while循环结束
}                                           // 主程序结束
```

8051互传实验（ch8–6.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图8–35连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

8.9 实时练习



在本章中探讨了串行式数据传输的概念、8051的串行端口、串行与并行数据的转换IC以及RS-232的驱动IC等。在此请试着回答下列问题，以确认对于此部分的掌握程度。

选择题

（ ）1. 下列哪片IC具有将串行数据转换成并行数据？
（A）74138 （B）74164 （C）74165 （D）74168

（ ）2. 下列哪片IC具有将并行数据转换成串行数据？
（A）74138 （B）74164 （C）74165 （D）74168

（ ）3. UART是指哪项装置？
（A）单向传输器 （B）万用串行数据与并行数据转换器
（C）全双工万用并行口 （D）万用异步串行端口

（ ）4. 同一个时间里，只能接收或传送信号者称为？
（A）半双工 （B）全双工 （C）半单工 （D）全单工

（ ）5. 在8051的串行口里，哪一种模式下，可利用Timer 1产生比特率？
（A）mode 0 （B）mode 1 （C）mode 2 （D）mode 3

（ ）6. 8051的串行口是通过哪些引脚进行数据传输？
（A）RxD引脚接收数据 （B）TxD引脚接收数据
（C）RxD传送数据 （D）以上皆非

（ ）7. 在8051里，若通过串行端口传出数据，则只要将数据放入哪个寄存器，CPU就会自动将它传出？
（A）SMOD （B）TBUF （C）SBUF （D）RBUF

（ ）8. 在8051里，若CPU完成串行端口数据的接收，将会如何？
（A）将TI标志位变为0 （B）将RI标志位变为0
（C）将TI标志位变为1 （D）将RI标志位变为1

（ ）9. 若要设定8051串行端口模式，可在哪个寄存器中设定？
（A）SMOD （B）SCON （C）PCON （D）TCON

（ ）10. 下列何者不是MAX232的功能？
（A）提升抗噪声能力 （B）提高传输距离 （C）增加传输速度 （D）以上皆是

问答题

1. 试写出8051串行端口4种工作模式的比特率及其设定方法。

2. 试说明8051串行口mode 2与mode 3的差异。

3. 当使用8051串行端口以mode 0工作模式将数据传出时，其TxD引脚与RxD引脚各担任何种任务？应如何接线？

4. 当使用8051串行端口，以mode 0工作模式接收数据时，其TxD引脚与RxD引脚各担任何种任务？应如何接线？

5. 试说明8051串行端口mode 1的数据格式。

6. 试说明8051的SCON寄存器中各位的功能。

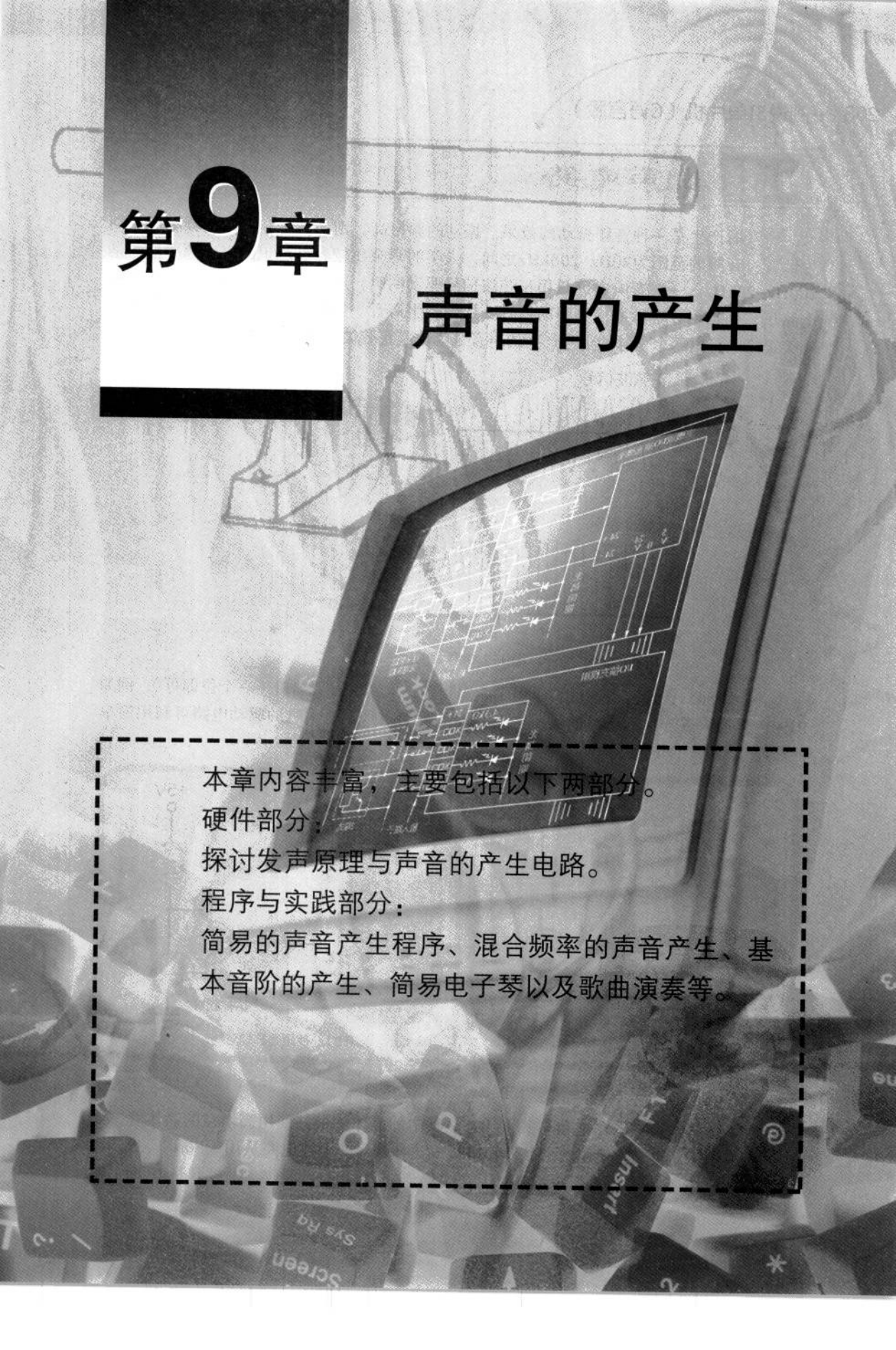
7. 试说明什么是“单工”、“半双工”与“全双工”。

8. 什么是“UART”？

9. 试简述如何应用74164及74165？

10. 试说明MAX232的功能？若要将8051系统的串行端口通过MAX232长距离传输，要如何连接？

第9章 声音的产生



本章内容丰富，主要包括以下两部分。

硬件部分：

探讨发声原理与声音的产生电路。

程序与实践部分：

简易的声音产生程序、混合频率的声音产生、基本音阶的产生、简易电子琴以及歌曲演奏等。

9.1 发声电路



声音的产生是一种音频振动的效果。振动的频率高，则为高音；振动的频率低，则为低音。音频的范围为20Hz~200kHz之间，人类的耳朵比较容易辨识的声音频率范围是200Hz~20kHz。一般音响电路是用正弦波信号驱动喇叭，从而产生悦耳的音乐；在数字电路里，则是用数字脉冲信号驱动喇叭，从而产生声音。如果声音的频率相同，人类的耳朵很难区分出哪个是脉冲信号产生的声音，哪个是正弦波信号产生的声音。

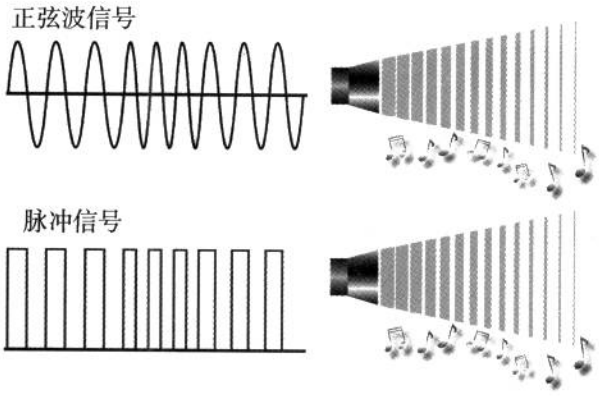


图9-1 声音的产生

若要用8051产生声音，可利用程序产生频率，送到输出入端口（一个位即可），例如P1.0，再把该点连接到喇叭的驱动电路，即可驱动喇叭。而喇叭的驱动电路可利用简单的晶体管电路，如图9-2所示。

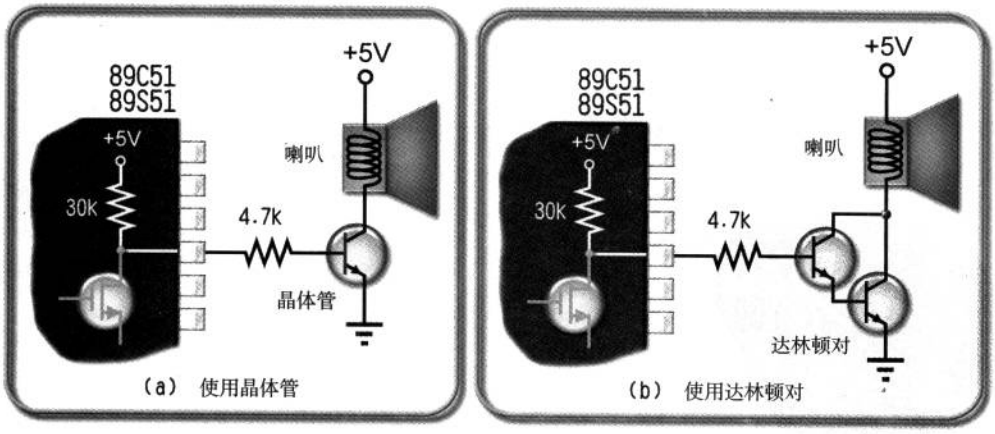


图9-2 喇叭驱动电路

为了让晶体管工作在饱和与截止状态，其中的晶体管最好使用高增益的晶体管，如达林顿晶体管，或干脆使用两个晶体管搭接成达林顿对。上述电路适用于Port 1、Port 2或Prot 3中的任意一个位，但是不要忘了，这3个输入/输出口都是类似漏极开路（OD）输

出，内部都有一个30kΩ的上拉电阻，其输出电流不可能太大。所以，在此所使用的R_B也不需要太大，1~5kΩ即可。

若要用Port 0作输出，就会麻烦一些！由于Port 0内部也是类似漏极开路输出，但它并没有上拉电阻，所以无法提供输出电流。因此，必须外接上拉电阻，如图9-3所示。

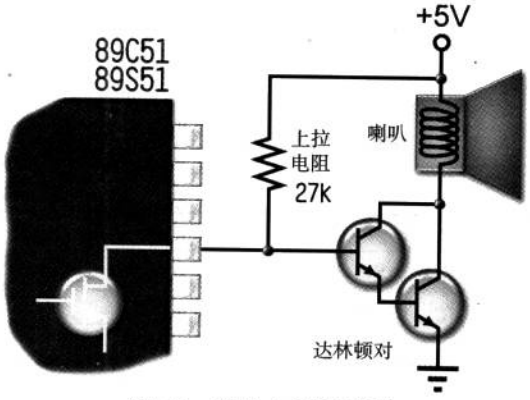


图9-3 由Port 0驱动喇叭

其中的电阻可以为20~30kΩ。

9.2 音调与节拍



在第3章中曾经介绍过蜂鸣器的驱动，只要在固定时间里切换输出的状态，即可让蜂鸣器（或喇叭）发出声响。当然，这个声响很单调，其音调的高低与输出的切换速度有关。切换速度越快，声音越高；反之，切换速度越慢，声音越低。除了控制发声的高低外，若还能控制发声的时间长短，这样就会有节奏感，也就形成了“音乐”的雏形。简单地讲，“音乐”至少要有声音的高低及发声的时间长短。

音调

若以频率来表示声音，有点抽象，又有点无趣！唱歌的时候，总不能跟乐队说“给我一个1kHz的Key”吧！通常是以Do、Re、Mi、Fa、So、La、Si、Do分别代表某一个频率的声音，我们称之为“音调”，即**Tone**。如表9-1所示为C调音阶表，包括3个音阶（低音、中音与高音），每个音阶为八音度，其中细分为12个半音（即Do、Do#、Re、Re#、Mi、Fa、Fa#、So、So#、La、La#、Si），而每个音阶之间的频率相差一倍，例如高音Do的频率（1046 Hz）刚好是中音Do的频率（523Hz）的一倍，中音Do的频率（523Hz）刚好是低音Do的频率（266Hz）的一倍；同样的，高音Re的频率（1 109Hz）刚好是中音Re的频率（554Hz）的一倍，中音Re的频率（554Hz）刚好是低音Re的频率（277Hz）的一倍，依次类推。因此，两个半音之间的频率比为$\sqrt[12]{2}$（大约是1.059），以中音为例，Do的频率为523Hz，所以Do#的频率为523 × 1.059，约为554Hz、Re的频率为554 × 1.059，约为587Hz……依次类推，如图9-4所示。

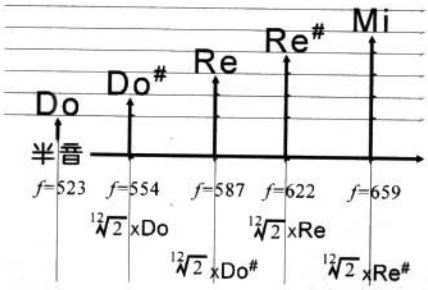


图9-4　音调与频率的关系

表9-1

音阶	n	1	2	3	4	5	6	7	8	9	10	11	12
		Do	Do#	Re	Re#	Mi	Fa	Fa#	So	So#	La	La#	Si
低音	频率	**262**	277	294	311	330	349	370	392	415	440	464	494
	简谱	1		2		3	4		5		6		7
中音	频率	**523**	554	587	622	659	698	740	784	831	880	932	988
	简谱	1		2		3	4		5		6		7
高音	频率	**1046**	1109	1175	1245	1318	1397	1480	1568	1661	1760	1865	1976
	简谱	1		2		3	4		5		6		7

以上音调相对于钢琴键盘的位置如图9-5所示。

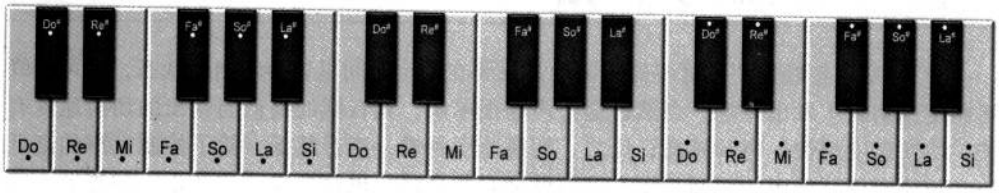


图9-5　钢琴键盘

节拍

若要构成音乐，光有音调是不够的！还需要节拍，让音乐具有旋律（固定的律动），而且可以调节各个音的快慢速度。“节拍”即**Beat**，简单说就是打拍子，例如听到音乐不自主地随之拍手或顿脚。我们常听说“这个音要1/4拍”、“那个音要1/2拍”，若1拍是0.5s，则1/4拍为0.125s、1/2拍为0.25s。至于1拍多少s，并没有严格规定，就像是人的心跳一样，大部分人的心跳是每分钟72下，有些人比较快，有些人比较慢，只要听顺耳就好。

除了“拍子”以外，还有“音节”，在乐谱左上方都会定义每个音节有多少拍，如图9-6所示。

以“生日快乐歌”的简谱为例，C 3/4代表为C调，4小节，每小节3拍；两条直线之间为一小节，其中有底线的两个1，代表一拍，之后的2、1各为一拍，总共3拍。在第2小节里，3后面的一条线代表3为两拍。以

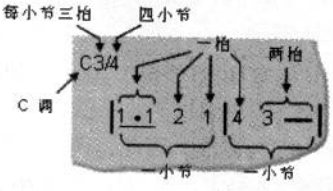


图9-6　音节

笔者“慢半拍”的习惯，唱一节的时间约2s，所以，每拍约3/2s。若以每小节1.5s的速度，可能会比较正常，也就是每拍0.5s、3/4拍0.375s、1/2拍0.25s、1/4拍0.125s……依次类推。若以程序来发出上述两小节的音，则是：

Do/0.25s、Do/0.25s、Re/0.5s、Do/0.5s

Fa/0.5s、Mi/1s 即

523Hz/250ms、523Hz/250ms、587Hz/500ms、523Hz/500ms

698Hz/500ms、659Hz/1000ms

9.3 音调的产生



若要产生表9-1所示的音频，可使用延迟函数或Timer定时中断，分别说明如下。

延迟函数

在Keil C里，使用延迟函数无法很精确地掌握所能延迟的时间长短，在前面的章节中，我们所使用的一个1ms延迟函数如下所示。

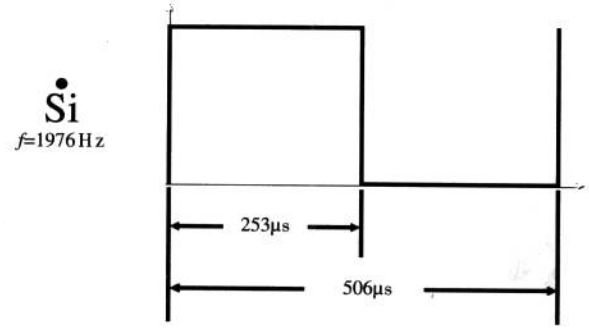
```
void delay1ms(unsigned char x)
{  unsigned char i,j;                    // 声明变量
   for(i=0;i<x;i++)                      // 外循环
         for(j=0;j<120;j++);             // 内循环
}
```

若要延迟1ms，则可使用“delay1ms（1）;”指令，则使函数的内循环将执行120次，外循环将执行1次。对于12MHz的8051系统而言，根据实验数据可得知，执行120次大约耗用1ms。同理，若需要延迟5ms，则可使用“delay1ms（5）;”。很明显地，这个函数的刻度为1ms，万一延迟时间要小于1ms怎么办？从上述函数中可得知，内循环的数量决定了延迟的时间，也就是说，若内循环的数量为120，可延迟1ms，该函数的最小刻度为1ms；若将内循环的数量降为12，可延迟0.1ms，该函数的最小刻度为0.1ms，即100μs。依次类推，整理出如表9-2所示的规律。

表9-2

内循环数	最小延迟时间（ms）	最小延迟时间（μs）
120	1	1000
60	0.5	500
6	0.05	50
3	0.025	25
1	0.0083	8.3

如表9-1所示，其中最高的音是高音的Si，其频率f=1976Hz，则周期T=506μs，半周期T1=253μs，即



若在延迟函数的内循环数量采用3，即

```
void delay25μs(unsigned char x)
{  unsigned char i,j;                    // 声明变量
   for(i=0;i<x;i++)                      // 外循环
         for(j=0;j<3;j++);               // 内循环
}
```

则“delay25μs（10）;”可产生250μs延迟，接近于高音Si的半周期。因此可产生高音Si。问题是，高音La*的频率为1865其周期为$\frac{1}{1865} \cong 536\,\mu s$，而半周期为268μs，若采用同一个延迟函数的话，到底要使用“delay25μs（10）;”指令，还是“delay25μs（11）;”指令呢？的确是个让人头痛的问题。明显地，以上函数的精度不够，所以改采用内循环为1的延迟函数，如下所示。

```
void delay8μs(unsigned char x)
{  unsigned char i,j;                    // 声明变量
   for(i=0;i<x;i++)                      // 外循环
         for(j=0;j<1;j++);               // 内循环
}
```

则高音Si可以由“delay8μs（30）;”指令产生，其中的参数30是253μ/8.33μ所得的值；而高音La*可以由“delay8μs（32）;”指令产生，其中的参数32是268μ/8.33μ所得的值。同理，以最低音（低音的Do）为例，其半周期为1908μs，可以由“delay8μs（229）;”指令产生。

根据上述方式推算，即可写出适用于delay8μs（int）函数的参数，如表9-3所示。

表9-3

低音	频率	T_1	参数	中音	频率	T_1	参数	高音	频率	T_1	参数
Do	262	**1908**	**229**	Do	523	**956**	**115**	Do	1046	**478**	**57**
Do#	277	**1805**	**217**	Do#	554	**903**	**108**	Do#	1109	**451**	**54**
Re	294	**1701**	**204**	Re	587	**852**	**102**	Re	1175	**426**	**51**
Re#	311	**1608**	**193**	Re#	622	**804**	**97**	Re#	1245	**402**	**48**
Mi	330	**1515**	**182**	Mi	659	**759**	**91**	Mi	1318	**379**	**45**
Fa	349	**1433**	**172**	Fa	698	**716**	**86**	Fa	1397	**358**	**43**
Fa#	370	**1351**	**162**	Fa#	740	**676**	**81**	Fa#	1480	**338**	**41**
So	392	**1276**	**153**	So	784	**638**	**77**	So	1568	**319**	**38**
So#	415	**1205**	**145**	So#	831	**602**	**72**	So#	1661	**301**	**36**
La	440	**1136**	**136**	La	880	**568**	**8**	La	1760	**284**	**34**
La#	464	**1078**	**129**	La#	932	**536**	**64**	La#	1865	**268**	**32**
Si	494	**1012**	**121**	Si	988	**506**	**61**	Si	1976	**253**	**30**

将这些音阶参数存入数组，如下所示。

```
unsigned char tone[3][12]={     {     229, 217, 204, 193, 182, 172,
                                      162, 153, 145, 136, 129, 121 },     //低音
                                {     115, 108, 102, 97, 91, 86,
                                      81, 77, 72, 68, 64, 61 },           // 中音
                                {     57, 54, 51, 193, 48, 45,
                                      41, 38, 36, 34, 32, 30 }};          // 高音
```

tone[0][x]为低八音阶，tone[1][x]为中八音阶，tone[2][x]为高八音阶若要取用音阶的参数，如下所示。



如要发出最低的音到最高的音，每个音重复执行50次，程序如下。

```
for(i=0;i<3;i++)                                    //从低八音到高八音
{    for(j=0;j<12;j++)                              //执行每个音阶
          for(k=0;k<50;k++)                         //每个音阶执行50次
          {     speaker=1;                          //输出高电平，speaker为输出端
                delay8μs(tone[i][j]);               //延迟
                speaker =0;                         //输出低电平
                delay8μs(tone[i][j]);               //延迟
          }                                         //结束一个音阶
```

```
}                                                    // 结束
```

上述方式可以发出所有音阶（36个音），不过有点复杂！在大部分的歌曲里，所使用的音域并不会那么宽，大多是在八音阶里，最多再增加个低音的Si、高音的Do与Re，至于中音的Do#、Re#、Fa#、So#、La#可以忽略。在一般的简谱中，常用的音阶编号如表9-4所示。

表9-4

简谱	音阶	频率	T_1	参数
0	Si	494	**1012**	**121**
1	Do	523	**903**	**108**
2	Re	587	**852**	**102**
3	Mi	659	**759**	**91**
4	Fa	698	**716**	**86**
5	So	784	**638**	**77**
6	La	880	**568**	**68**
7	Si	988	**506**	**61**
8	Do	1046	**478**	**57**
9	Re	1175	**426**	**51**
10	Mi	1318	**379**	**45**

在此将低音的Si编码为0，高音的Do编码为8，高音的Re编码为9，高音的Mi编码为10，将这些数据放入数组，如下所示。

```
unsigned char tone[11]= {   121, 108, 102, 91, 86, 77,
                            68, 61, 57, 51, 45 };
```

如此一来，所有事情都简单了！如果使用一般的简谱，只要把低音的Si改为0，把高音的Do、Re、Mi改为8、9、10即可。以“我是只小小鸟”为例，其简谱如下所示。

```
C调  3/6                我是一只小小鸟
| 1  1  1  1 | 3· 2  1 | 3  3  3 | 5· 4  3 | 5  4  3 | 2 - - |
  我是只  小小鸟，飞就飞，叫就叫，自由逍遥。
| 2 - 1 [7] | 1  2  3 | 4 - 3  2 | 3  4  5 | 5  4  3  2 | 1 - - |
  我  不知  有忧愁，我不知  有烦恼，只是常欢  笑。
```

则可设置一个歌谱的数组，如下所示为第一小段的歌谱数组。

```
unsigned char song[]=   {       1, 1, 1,  3, 2, 1,  3, 3, 3,  5, 4, 3,  5, 4, 3,  2};
```

当我们要演奏时，则可利用此数组与刚才的tone数组进行转码与输出，如下所示。

```
unsigned char  i,j;                                    // 声明变量
unsigned char song[]=    {   1, 1, 1,  3, 2, 1,  3, 3, 3,  5, 4, 3,  5, 4, 3,  2 };
unsigned char tone[11]= {   121, 108, 102, 91, 86, 77,
                            68, 61, 57, 51, 45 }
  for(i=0;i<16;i++)                                    // 读取16个音
  {       for(j=0;j<50;j++)                            // 每个音阶执行50次
          {   speaker=1;                               // 输出高电平
              delay8μs(tone[song[i]]);                 // 延迟
              speaker=0;                               // 输出低电平
              delay8μs(tone[song[i]]);                 // 延迟
          }                                            // 结束一个音阶
}                                                      // 结束
```

若要改变演奏的歌时，只要将歌的简谱填入song[]数组即可。

定时中断

以定时器中断产生频率的方式在第7章中已经讲过了，声音只是某一范围的频率，也就是所谓的音频。在Mode 1模式下，定时量最多可达65 536，也就是65 536μs，足以产生低音Do所需的半周期（1908）。所以，若要产生低音Do的音频，则只需要执行1908定时量的Timer中断即可。每中断一次，就改变连接喇叭的输入/输出口的状态，就能发出低音Do的声音。若要产生其他音阶，只需要按表9-3的$\mathbf{T_1}$字段设定定时量即可。如下所示的程序以Mode 1来产生低音的Do。

```
#include  <reg51.h>
sbit    speaker = P1^0;                        // 声明输出端
main()
{ speaker=0;                                   // 喇叭初始值
  IE=0x82;                                     // 启用Timer 0
  TH0=(65536-1908)/256;                        // 填入定时量的高八位
  TL0=(65536-1908)%256;                        // 填入定时量的低八位
  TR0=1;                                       // 启动Timer 0
  while(1);                                    // 停止
  }                                            // 主程序结束
//====Timer 0中断子程序==================
void tone_int(void)interrupt 1                 // Timer 0中断子程序开始
{ TH0=Do_H;                                    // 填入定时量的高八位
```

```
    TL0=Do_L;                          // 填入定时量的低八位
    speaker=~speaker;                  // 喇叭反相输出
}                                      // 结束中断子程序
```

上述程序只能用来播放一个音，若要播放一串音，则可利用刚才的tone数组，进行转码与输出，如下所示。

```
#include  <reg51.h>
sbit   speaker = P1^0;                          // 声明输出端
unsigned char   i;                              // 声明变量
unsigned char tone_H, tone_L;                   // 声明定时量变量
unsigned char song[]=   {  1, 1, 1,  3, 2, 1,  3, 3, 3,  5, 4, 3,  5, 4, 3,  2};
unsigned char tone[11]= {  1012, 903, 852, 759, 716, 638,
                           568, 506, 478, 426, 379 };
main()
{ speaker=0;                                    // 喇叭初始值
  IE=0x82;                                      // 启用Timer 0
  for(i=0;i<16;i++)                             // 读取16个音
        {  tone_H=(65536-tone[song[i]])/256;
           /*读取音阶计数量的高八位*/
           tone_L=(65536-tone[song[i]])%256;
           /*读取音阶计数量的低八位*/
           TH0=tone_H;                          // 填入音阶计数量的高八位
           TL0=tone_L;                          // 填入音阶计数量的低八位
           TR0=1;                               // 启动Timer 0
           delay1ms(100);                       // 每个音播放0.1s
        }                                       // 结束一个音阶
 }                                              // 结束播放13个音
}                                               // 主程序结束
//====Timer 0中断子程序==================
void tone_timer(void)interrupt 1                // Timer 0中断子程序开始
{ TH0=tone_H;                                   // 填入定时量的高八位
  TL0=tone_L;                                   // 填入定时量的低八位
  speaker=~speaker;                             // 喇叭反相输出
}                                               // 结束中断子程序
```

上述播放16个音，事先要知道所要播放的有多少个音有点麻烦！如果在数组的最后一个位置，放置一个15作为结束符号，我们就可利用这个15来判断是否要结束播放，说

明如下。

```
#include <reg51.h>
sbit   speaker = P1^0;                          // 声明输出端
unsigned char   i=0;                            // 声明变量
unsigned char tone_H, tone_L;                   // 声明定时量变量
unsigned char song[]=   {  1, 1, 1,  3, 2, 1,  3, 3, 3,  5, 4, 3,  5, 4, 3,  2,  15 };
unsigned char tone[11]= {  1012, 903, 852, 759, 716, 638,
                           568, 506, 478, 426, 379 };
main()
{ speaker=0;                                    // 喇叭初始值
  IE=0x82;                                      // 启用Timer 0
  while(song[i]!=15)                            // while循环开始
  {     tone_H=(65536-tone[song[i]])/256;       // 读取音阶计数量的高八位
        tone_L=(65536-tone[song[i]])%256;       // 读取音阶计数量的低八位
        TH0=tone_H;                             // 填入音阶计数量的高八位
        TL0=tone_L;                             // 填入音阶计数量的低八位
        TR0=1;                                  // 启动Timer 0
        delay1ms(100); // 每个音播放0.1s
        i++;                                    // 下一个音
  }                                             // 结束播放
}                                               // 主程序结束
//====Timer 0中断子程序==================
void tone_timer(void)interrupt 1                // Timer 0中断子程序开始
{ TH0=tone_H;                                   // 填入定时量的高八位
  TL0=tone_L;                                   // 填入定时量的低八位
  speaker=~speaker;                             // 喇叭反相输出
}                                               // 结束中断子程序
```

9.4 节拍的产生



音阶的频率是固定的，而节拍有快有慢，拍子越短，节奏越快，拍子越长，节奏越慢。产生节拍的方法也是一种处理时间的方法。以生日快乐歌的前两个音节为例，第一个音是Do，发生这个音的时间长度是250ms（也就是在250ms之内都在产生Do的音）；停顿一下，再发出第二个音（Do），还是持续250ms；接下来，改发出Re的音，时间长达500ms、改发出Do的音，时间长达500ms，第一小节结束。紧接着第二小节，首先发出Fa

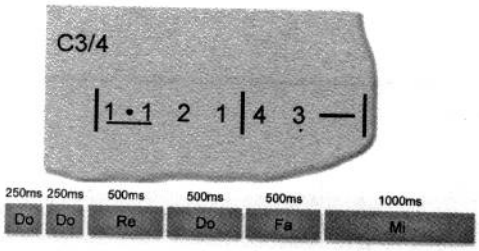


图9-7 节拍

的音，时间长达500ms；再发出Mi的音，时间长达1000ms……如图9-7所示。因此，必须把其中的节拍转换成节拍数组，以刚才讲到的“我是一只小小鸟”的简谱为例，整个节拍说明如下：

|1、1、1|1.5、0.5、1|1、1、1|1.5、0.5、1|1、1、1|3|

|2、0.5、0.5|1、1、1|2、0.5、0.5|1、1、1|0.5、0.5、1、1|3|

如何控制发音的时间呢？同样是延迟函数或Timer定时中断两种方式，说明如下。

● 延迟函数

若音阶产生的方式是Timer中断的方式，则节拍产生的方式就可采用延迟函数的方式。首先整理出整首乐曲中的拍子种类，找出其中最短的拍子，例如整首乐曲之中，包含1/4拍、1/2拍、3/4拍、1拍及2拍，则以1/4拍为基准，然后写一段1/4拍长度的延迟函数，若要产生1/4拍的长度，则执行该函数时，自变量为1；若要产生1/2拍的长度，则执行该函数时，自变量为2；若要产生3/4拍的长度，则执行该函数时，自变量为3；若要产生1拍的长度，则执行该函数时，自变量为4；若要产生2拍的长度，则执行该函数时，自变量为8……依次类推。若1/4拍的长度为0.125s（即125ms），则beat_125（）函数，说明如下。

```
void beat_125(unsigned char x)                    // 节拍函数开始
{ unsigned char i,j,k;                            // 声明变量
  for(i=0;i<x;i++)                                // i循环
        for(j=0;j<125;j++)                        // j循环
            for(k=0;i<120;k++);                   // k循环
}                                                 // 结束节拍函数
```

如上的程序，若要延迟1拍，则自变量为4，使用如下指令。

```
beat_125(4);                                      // 1拍
```

因此，如果利用beat_125（）函数产生节拍，则“我是只小小鸟”的节拍数组可定义如下。

```
unsigned char beat[11]= {      4, 4, 4,  6, 2, 4,  4, 4, 4,  6, 2, 4,  4, 4, 4,  12,
                               8, 2, 2,  4, 4, 4,  8, 2, 2  4, 4, 4,  2, 2, 4, 4,  12 };
```

若要使用Timer中断方式产生音阶，使用延迟函数的方式产生节拍，如下所示。

```
#include <reg51.h>
sbit  speaker = P1^0;                                  // 声明输出端
unsigned char  i=0;                                    // 声明变量
unsigned char  tone_H, tone_L;                         // 声明定时量变量
void  beat_125(unsigned char);                         // 声明节拍函数
unsigned char song[]=    {  1, 1, 1,  3, 2, 1,  3, 3, 3,  5, 4, 3,  5, 4, 3,  2,
                            2, 1, 0,  1, 2, 3,  4, 3, 2,  3, 4, 5,  5, 4, 3, 2,  1, 15 };
unsigned char beat[]=    {  4, 4, 4,  6, 2, 4,  4, 4, 4,  6, 2, 4,  4, 4, 4,  12,
                            8, 2, 2,  4, 4, 4,  8, 2, 2  4, 4, 4,  2, 2, 4, 4,  12 };
unsigned char tone[11]= {  1012, 903, 852, 759, 716, 638,
                            568, 506, 478, 426, 379 };
main()
{ speaker=0;                                           // 喇叭初始值
  IE=0x82;                                             // 启用Timer 0
 while(song[i]!=15)                                    // while循环开始
 {      tone_H=(65536-tone[song[i]])/256;              // 读取音阶计数量的高八位
        tone_L=(65536-tone[song[i]])%256;              // 读取音阶计数量的低八位
        TH0=tone_H;                                    // 填入音阶计数量的高八位
        TL0=tone_L;                                    // 填入音阶计数量的低八位
        TR0=1;                                         // 启动Timer 0
        beat_125(beat[i]);                             // 指定节拍
        i++;                                           // 下一个音
  }                                                    // 结束播放
}                                                      // 主程序结束
//====Timer 0中断子程序==================
void tone_timer(void)interrupt 1                       // Timer 0中断子程序开始
{ TH0=tone_H;                                          // 填入定时量的高八位
  TL0=tone_L;                                          // 填入定时量的低八位
  speaker=~speaker;                                    // 喇叭反相输出
}                                                      // 结束中断子程序
//=====节拍函数==================
void beat_125(unsigned char x)                         // 节拍函数开始
{ unsigned char i,j,k;                                 // 声明变量
```

```
    for(i=0;i<x;i++)                    // i循环
        for(j=0;j<125;j++)              // j循环
            for(k=0;k<120;k++);         // k循环
}                                       // 结束节拍函数
```

定时中断

不管音阶产生的方式是Timer中断方式，还是延迟函数的方式，其节拍都可利用定时中断产生。同样是找出整首乐曲中最短的拍子，例如整首乐曲之中，最短是1/4拍，若1/4拍的时间为0.125s，则以1/4拍为基准，然后设定每0.125s产生一次中断，其定时量为125000。若采用mode 1，而定时量设为62500，则只要执行2次中断，即可产生1/4拍的时间长度。同样地，若要产生1/2拍的长度，则执行4次中断、若要产生3/4拍的长度，则执行6次中断……依次类推，如表9-5所示。

表9-5

拍数	中断次数	拍数	中断次数	拍数	中断次数
1/8	1	1/2	4	1又1/4	10
1/4	2	3/4	6	1又1/2	12
3/8	3	1	8	2	16

以下程序是利用延迟函数产生音阶，以定时器中断的方式（Mode 1）产生节拍。其动作是发出Do的音阶1/4拍，然后静音1/4拍，如此循环5次。在此以1/8拍的定时量为基础，times为中断次数，也就是1/8拍的倍数，另外设置节拍标志位beat_flag，若beat_flag=1，表示拍子结束。

```
#include <reg51.h>
sbit  speaker=P1^0;                          // 声明speaker位置
unsigned char Do=108;                        // 声明Do变量
unsigned char beat_H=(65536-62500)/256;      // 声明节拍定时量的高八位
unsigned char beat_L=(65536-62500)%256;      // 声明节拍定时量的低八位
char times;                                  // 声明节拍重复次数
char counts=5;                               // 声明循环次数
bit beat_flag;                               // 声明节拍标志位
main()
{ char i;                                    // 声明变量
  IE=0x82;                                   // 启用Timer 0中断
  TH0=beat_H;                                // 填入定时量的高八位
  TL0=beat_L;                                // 填入定时量的低八位
  for(i=0;i<counts;i++)
```

```
  {     times=2;                                  // 设定重复次数
        beat_flag=0;                              // 设定节拍标志位
        TR0=1;                                    // 启动Timer 0
//==发Do音(1/4拍)============================
        if(beat_flag==0)
        {  speaker=~speaker;                      // 切换输出状态
           delay8μs(tone[Do]);                    // 延迟
        }                                         // 结束一个音阶
//==静音(1/4拍)==============================
        speaker=0;                                // 关闭输出
        times=2;                                  // 设定重复次数
        beat_flag=0;                              // 设定节拍标志位
        TR0=1;                                    // 启动Timer 0
        while(beat_flag=0);                       // 停止
  }                                               // 循环结束
}                                                 // 主程序结束
//=====================================
void beat_timer(void)interrupt 1
{ TH0=beat_H;                                     // 填入定时量的高八位
  TL0=beat_L;                                     // 填入定时量的低八位
  if(--times==0)                                  // 判断次数到了吗?
        { beat_flag=1;                            // 切换标志位
          TR0=0; }                                // 关闭Timer 0
}
//=====================================
void delay8μs(unsigned char x)
{ unsigned char i,j;                              // 声明变量
  for(i=0;i<x;i++)                                // 外循环
        for(j=0;j<1;j++);                         // 内循环
}
```

紧接着是利用Timer 0产生音阶，Timer 1产生节拍，而以定时器中断的方式（Mode 1）产生节拍，同样是发出Do的音阶1/4拍，然后静音1/4拍，如此循环5次，程序如下所示。

```
#include <reg51.h>
sbit  speaker=P1^0;                               // 声明speaker位置
unsigned char tone_H=(65536-903)/256;             // 声明音阶定时量的高八位
```

```
unsigned char tone_L=(65536-903)%256;          // 声明音阶定时量的低八位
unsigned char beat_H=(65536-62500)/256;        // 声明节拍定时量的高八位
unsigned char beat_L=(65536-62500)%256;        // 声明节拍定时量的低八位
char times;                                    // 声明节拍重复次数
char counts=5;                                 // 声明循环次数
bit beat_flag;                                 // 声明节拍标志位
main()
{ char i;                                      // 声明变量
  IE=0x8a;                                     // 启用Timer 0/1中断
  TH0=tone_H;                                  // 填入音阶定时量的高八位
  TL0=tone_L;                                  // 填入音阶定时量的低八位
  TH1=beat_H;                                  // 填入节拍定时量的高八位
  TL1=beat_L;                                  // 填入节拍定时量的低八位
  for(i=0;i<counts;i++)
  {     times=2;                               // 设定重复次数(1/4拍)
        beat_flag=0;                           // 设定节拍标志位
        TR0=1;                                 // 启动Timer 0
        TR1=1;                                 // 启动Timer 1
//==发Do音(1/4拍)==========================
        while(beat_flag==0);                   // 停止
//==静音(1/4拍)============================
        speaker=0;                             // 关闭输出
        times=2;                               // 设定重复次数(1/4拍)
        beat_flag=0;                           // 设定节拍标志位
        TR0=1;                                 // 启动Timer 0
        while(beat_flag=0);                    // 停止
  }                                            // 循环结束
}                                              // 主程序结束
//===音阶中断==================================
void beat_timer(void)interrupt 1
{ TH0=tone_H;                                  // 填入音阶定时量的高八位
  TL0=tone_L;                                  // 填入音阶定时量的低八位
  if(beat_flag==1)TR0=0;                       // 标志位等于1，则停止发音
  else speaker=~speaker;                       // 切换输出状态
}
//===节拍中断==================================
void beat_timer(void)interrupt 3
{ TH1=beat_H;                                  // 填入节拍定时量的高八位
```

```
    TL1=beat_L;                                    // 填入节拍定时量的低八位
    if(--times==0)                                 // 判断次数到了吗?
          {  beat_flag=1;                          // 切换标志位
             TR1=0; }                              // 关闭Timer 0
}
```

如果利用beat_timer中断子程序产生节拍，则“我是只小小鸟”的节拍数组可定义如下。

```
unsigned char beat[11]= {  8, 8, 8,  12, 4, 8,  8, 8, 8,  12, 4, 8,  8, 8, 8,  24,
                           16, 4, 4,  8, 8, 8,  16, 4, 4  8, 8, 8,  4, 4, 8, 8,  24 };
```

若要使用Timer中断方式产生音阶及节拍，程序如下所示。

```
sbit  speaker=P1^0;                                // 声明speaker位置
unsigned char  i=0;                                // 声明变量
unsigned char tone_H;                              // 声明音阶定时量的高八位
unsigned char tone_L;                              // 声明音阶定时量的低八位
unsigned char beat_H=(65536-62500)/256;            // 声明节拍定时量的高八位
unsigned char beat_L=(65536-62500)%256;            // 声明节拍定时量的低八位
char times;                                        // 声明节拍重复次数
bit beat_flag;                                     // 声明节拍标志位
unsigned char beat[]=     {  8, 8, 8,  12, 4, 8,  8, 8, 8,  12, 4, 8,  8, 8, 8,  24,
                             16, 4, 4,  8, 8, 8,  16, 4, 4  8, 8, 8,  4, 4, 8, 8,  24 };
unsigned char song[]=     {  1, 1, 1,  3, 2, 1,  3, 3, 3,  5, 4, 3,  5, 4, 3,  2,
                             2, 1, 0,  1, 2, 3,  4, 3, 2,  3, 4, 5,  5, 4, 3, 2,  1, 15 };
unsigned char tone[11]= {  1012, 903, 852, 759, 716, 638,
                           568, 506, 478, 426, 379 };
main()
{ IE=0x8a;                                         // 启用Timer 0/1中断
  while(song[i]!=15)
  {       tone_H=(65536-tone[song[i]])/256;        // 读取音阶定时量的高八位
          tone_L=(65536-tone[song[i]])%256;        // 读取音阶定时量的高八位
          TH0=tone_H;                              // 填入音阶定时量的高八位
          TL0=tone_L;                              // 填入音阶定时量的高八位
          times=beat[i];                           // 读取节拍的重复次数
          TH1=beat_H;                              // 填入节拍定时量的高八位
          TL1=beat_L;                              // 填入节拍定时量的低八位
          beat_flag=0;                             // 设定节拍标志位
```

```
        TR0=1;                                  // 启动Timer 0
        TR1=1;                                  // 启动Timer 1
        while(beat_flag==0);                    // 停止
        i++;                                    // 下一个音
    }                                           // 循环结束
}                                               // 主程序结束
//===音阶中断==================================
void beat_timer(void)interrupt 1
{ TH0=tone_H;                                   // 填入音阶定时量的高八位
  TL0=tone_L;                                   // 填入音阶定时量的低八位
  if(beat_flag==1)TR0=0;                        // 标志位等于1，则停止发音
  else speaker=~speaker;                        // 切换输出状态
}
//===节拍中断==================================
void beat_timer(void)interrupt 3
{ TH1=beat_H;                                   // 填入节拍定时量的高八位
  TL1=beat_L;                                   // 填入节拍定时量的低八位
  if(--times==0)                                // 判断次数到了吗?
          { beat_flag=1;                        // 切换标志位
            TR1=0; }                            // 关闭Timer 0
}
```

音阶及节拍的处理方式分别可以采用延迟函数及定时器中断的方式，在它们的组合应用上有如下特点。

- 使用延迟函数产生音阶及节拍：节拍不会准确。
- 使用延迟函数产生音阶，使用定时器中断产生节拍：音阶与节拍都可以比较准确，但程序不能做其他事情。
- 使用延迟函数产生节拍，使用定时器中断产生音阶：音阶与节拍都可以比较准确，但程序不能做其他事情。
- 使用定时器中断产生产生音阶及节拍：音阶与节拍都可以很准确，同时程序还能处理其他事情。

9.5 实例演练



在本小节中提供4个实例，以介绍8051的发声方法，如下所示。

9.5.1 简易电子琴实例演练

实验要点

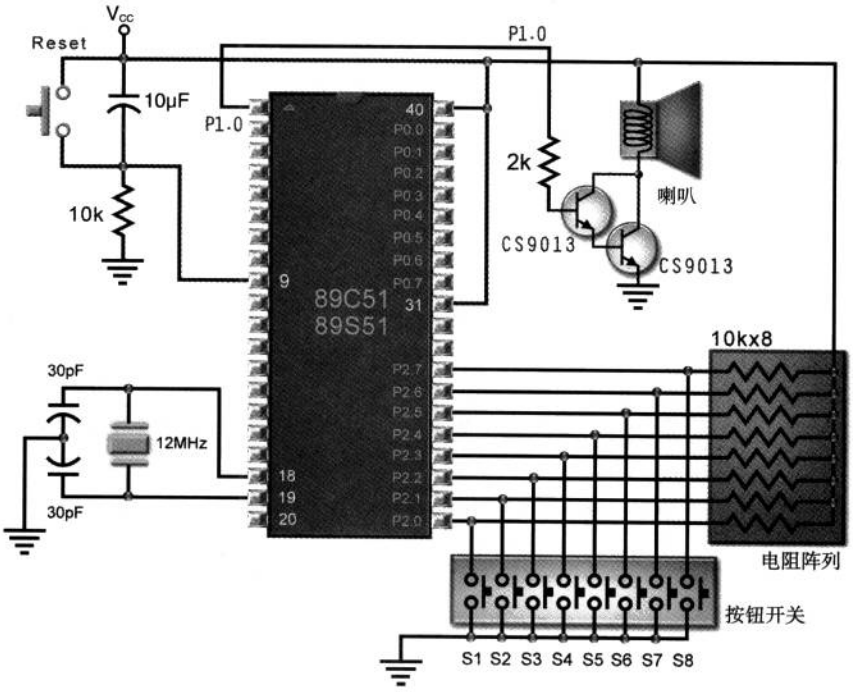


图9-8 简易电子琴电路图

如图9-8所示，根据功能需求与电路结构得知，当按钮开关on时，将可由其连接的输入口读取到低电平（即0）。在此要制作一个八键的电子琴，若按S1，则发出中音的Do，若按S2，则发出中音的Re……依次类推。

流程图与程序设计

根据表9-3中的参数字段可以写出一个按键与参数的对照表，如表9-6所示。

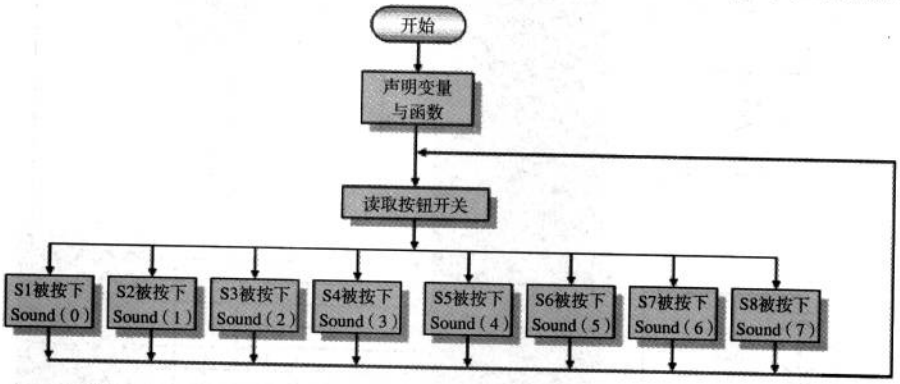
表9-6

按键	音阶	参数
S1	中音Do	108
S2	中音Re	102
S3	中音Mi	91
S4	中音Fa	86
S5	中音So	77
S6	中音La	68
S7	中音Si	61
S8	高音Do	57

根据这个对照表，可以列出一个tone数组，如下所示。

```
unsigned char tone[]=   {   108, 102, 91, 86, 77, 68, 61, 57 }
```

我们将根据这个数组，将按钮开关状态转换成延迟函数的参数。而按钮开关状态的判断，可以使用switch-case语句来判断按钮开关状态，相当方便！整个流程图与程序如下所示。




```
/* ch9-1.c - 简易电子琴实验 */
#include <reg51.h>                                  // 包含reg51.h文件
sbit speaker=P1^0;                                  // 声明喇叭位置
unsigned char keys;                                 // 声明变量
/* 声明音阶数组 --    Do  Re  Mi  Fa  So  La  Si  Do' */
unsigned char tone[]= { 108, 102, 91, 86, 77, 68, 61, 57 };
void sound(unsigned char);                          // 声明发声函数
void delay8μs(unsigned char);                       // 声明延迟函数
//====主程序=====================================
main()                                              // 主程序开始
{ P2=0xff;                                          // 将Port 2规划成输入口
 while(1)                                           // while循环
 {      keys=~P2;                                   // 读取按钮
        switch(keys)                                // 判断
        {   case 0x01: sound(0); break;             // 按下S1，发Do音
            case 0x02: sound(1); break;             // 按下S2，发Re音
            case 0x04: sound(2); break;             // 按下S3，发Mi音
            case 0x08: sound(3); break;             // 按下S4，发Fa音
            case 0x10: sound(4); break;             // 按下S5，发So音
            case 0x20: sound(5); break;             // 按下S6，发La音
            case 0x40: sound(6); break;             // 按下S7，发Si音
```

```
            case 0x80: sound(7); break;                    // 按下S8，发Do#音
        }
    }                                                      // while循环结束
}                                                          // 主程序结束
//=====发声函数==============================
void sound(unsigned char x)                                // 发声函数开始
{ char i;                                                  // 声明变量
  for(i=0;i<60;i++)                                        // 执行60次
  {   speaker=1;                                           // 输出高电平
      delay8μs(tone[x]);                                   // 延迟
      speaker=0;                                           // 输出低电平
      delay8μs(tone[x]);                                   // 延迟
  }                                                        // 结束一个音
}                                                          // 结束
//======延迟函数==============================
void delay8μs(unsigned char x)                             // 延迟函数开始
{ unsigned char i,j;                                       // 声明变量
  for(i=0;i<x;i++)                                         // 外循环
      for(j=0;j<1;j++);                                    // 内循环
}                                                          // 结束
```

简易电子琴实验（ch9-1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图9-7连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若有非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

思考一下

1. 本实验里，有无抖动的困扰？

2. 请改用定时器方式，重新设计此电子琴？

3. 在本实验里，若要改由P1.7输出音频，程序应如何修改？

9.5.2 DoReMi实例演练

实验要点

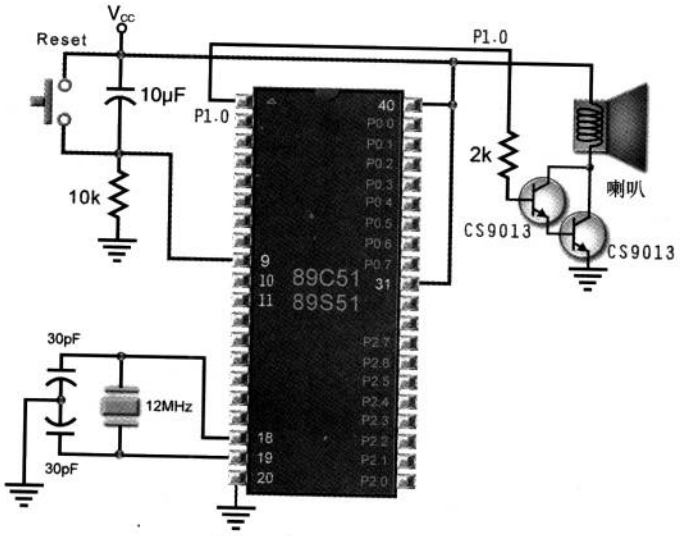


图9-9 DoReMi电路图

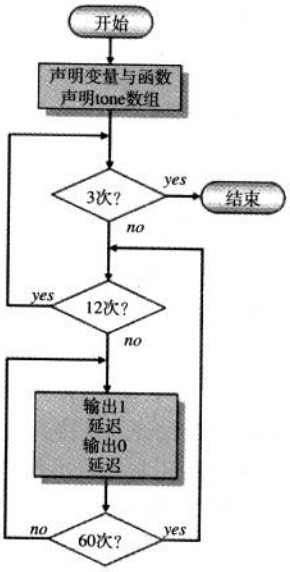
如图9-9所示的电路图，每隔一小段时间发一个音，从低音的Do开始直到高音的Si，总共36个音阶。

流程图与程序设计

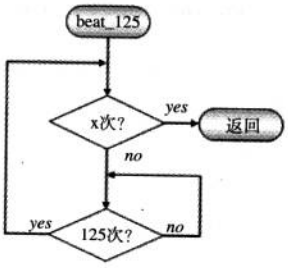
根据表9-3中的参数，将程序中延迟函数的参数放入一个二维数组，如下所示。

```
unsigned char tone[3][12]={   {  229, 217, 204, 193, 182, 172,
                                 162, 153, 145, 136, 129, 121 },   // 低音
                              {  115, 108, 102, 97, 91, 86,
                                 81, 77, 72, 68, 64, 61 },         // 中音
                              {  57, 54, 51, 193, 48, 45,
                                 41, 38, 36, 34, 32, 30 }};        // 高音
```

在此利用三层的for语句，依次读取数组中的数据，也就是音阶的参数，作为delay8μs（）函数的自变量，而每个音重复执行60次，整个流程图与程序如下所示。



主程序




```
/* ch9-2.c - DoReMi实验 */
#include  <reg51.h>
sbit  speaker=P1^0;                          // 声明喇叭位置
unsigned char tone[3][12]={   {  229, 217, 204, 193, 182, 172,
                                 162, 153, 145, 136, 129, 121 },      //低音
                              {  115, 108, 102, 97, 91, 86,
                                 81, 77, 72, 68, 64, 61 },            //中音
                              {  57, 54, 51, 193, 48, 45,
                                 41, 38, 36, 34, 32, 30 }};           //高音
void delay8μs(unsigned char);                // 声明延迟函数
//================================================
main()                                       // 主程序开始
{ char i,j,k;                                // 声明变量
  for(i=0;i<3;i++)                           // 从低八音到高八音
  {     for(j=0;j<12;j++)                    // 执行每个音阶
              for(k=0;k<60;k++)              // 每个音阶执行60次
              {     speaker=1;               // 输出高电平，speaker为输出端
                    delay8μs(tone[i][j]);    // 延迟
                    speaker =0;              // 输出低电平
                    delay8μs(tone[i][j]);    // 延迟
              }                              // 结束一个音阶
  }                                          // 结束
```

```
}
//=======================================
void delay8μs（unsigned char x）        // 延迟函数开始
{ unsigned char i,j;                    // 声明变量
  for（i=0;i<x;i++）                    // 外循环
        for（j=0;j<1;j++）;             // 内循环
}                                       // 结束
```

DoReMi实验（ch9-2.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图9-8连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

思考一下

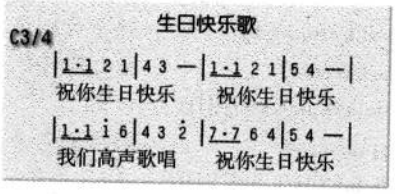
1. 如果要以定时器来完成本实验的功能，程序应如何编写？

2. 若设置一个按钮开关，每按一下就依次发出这36个音，电路应如何修改？程序应如何编写？

9.5.3 生日快乐歌实例演练

实验要点

同9.5.2节的电路图（见图9-8），在此要演奏“生日快乐歌”，如下所示为“生日快乐歌”的简谱。



流程图与程序设计

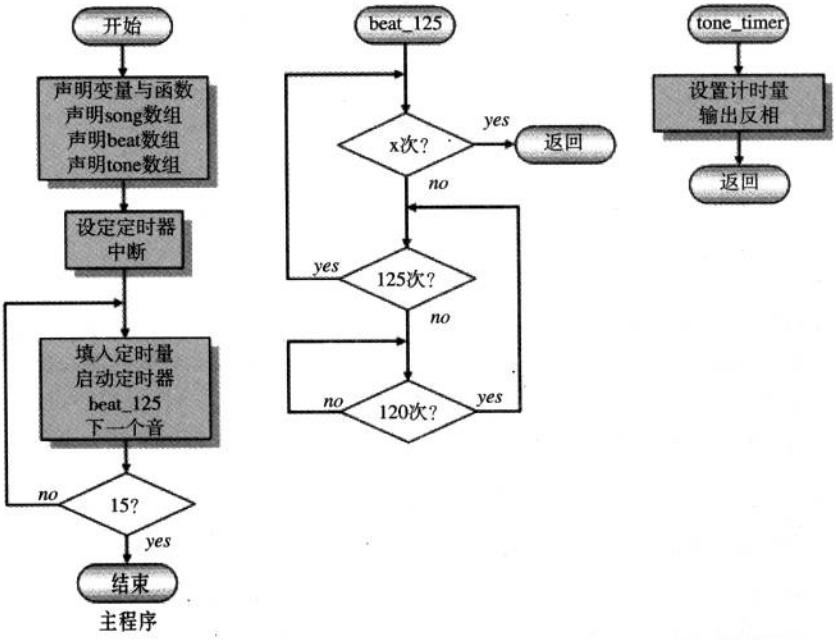
首先将简谱的音阶存入song[]数组，而该数组的最后放置15，如下所示。

```
unsigned char song[]={  1, 1, 2, 1,  4, 3,  1, 1, 2, 1,  5, 4,
                        1, 1, 8, 6,  4, 3, 9,  7, 7, 6, 4,  5, 4,  15};
```

其中比较特殊的 $\frac{g}{1}$ 及 $\frac{g}{2}$，$\frac{g}{1}$ 为高音Do，放在为数组中的song[8]；2为高音Re，放在为数组中的song[9]。紧接着，根据表9-5将简谱的节拍存入beat[]数组，如下所示。

```
unsigned char beat[]={  4, 4, 8, 8,  8, 16,  4, 4, 8, 8,  8, 16,
                        4, 4, 8, 8,  8, 8, 8,  4, 4, 8, 8,  8, 16};
```

根据上述数组及基本的音阶数组（tone[]），采用定时器中断方式产生音阶，采用延迟函数方式产生节拍，完整流程图与程序设计如下所示。



主程序

```
/* 生日快乐歌实验(ch9-3.c)*/
#include <reg51.h>
sbit  speaker = P1^0;                          // 声明输出端
unsigned char  i=0;                            // 声明变量
unsigned char  tone_H, tone_L;                 // 声明定时量变量
void  beat_125(unsigned char);                 // 声明节拍函数
unsigned char song[]={  1, 1, 2, 1,  4, 3,  1, 1, 2, 1,  5, 4,
                        1, 1, 8, 6,  4, 3, 9,  7, 7, 6, 4,  5, 4,  15};  // 歌曲
```

```
unsigned char beat[]={   4, 4, 8, 8,  8, 16,  4, 4, 8, 8,  8, 16 ,
                         4, 4, 8, 8,  8, 8, 8,  4, 4, 8, 8,  8, 16};      // 节拍
unsigned char tone[]={   1012, 903, 852, 759, 716, 638,
                         568, 506, 478, 426, 379 };                        // 音阶定义
// ========主程序 ================
main()
{ speaker=0;                                          // 喇叭初始值
  EA=1;                                               // 开启中断总开关
  ET0=1;                                              // 开启Timer 0中断开关
  while(song[i]!=15)                                  // while循环开始
  {     tone_H=(65536-tone[song[i]])/256;             // 读取音阶计数量的高八位
        tone_L=(65536-tone[song[i]])%256;             // 读取音阶计数量的低八位
        TH0=tone_H;                                   // 填入音阶计数量的高八位
        TL0=tone_L;                                   // 填入音阶计数量的低八位
        TR0=1;                                        // 启动Timer 0
        beat_125(beat[i]);                            // 指定节拍
                i++;                                  // 下一个音
  }                                                   // 结束播放
}                                                     // 主程序结束
//====Timer 0中断子程序==================
void tone_timer(void)interrupt 1                      // Timer 0中断子程序开始
{ TH0=tone_H;                                         // 填入定时量的高八位
  TL0=tone_L;                                         // 填入定时量的低八位
  speaker=~speaker;                                   // 喇叭反相输出
}                                                     // 结束中断子程序
//====节拍函数==================
void beat_125(unsigned char x)                        // 节拍函数开始
{ unsigned char i,j,k;                                // 声明变量
  for(i=0;i<x;i++)                                    // i循环
      for(j=0;j<125;j++)                              // j循环
          for(k=0;k<120;k++);                         // k循环
}                                                     // 结束节拍函数
```

生日快乐歌实验（ch9-3.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状

况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图9-8连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

思考一下

1. 如果要以两个定时器来完成本实验的功能，程序应如何编写？

2. 若设置一个按钮开关，每按一下演奏一次“生日快乐歌”，电路应如何修改？程序应如何编写？

9.5.4 快乐点唱机实例演练一

实验要点

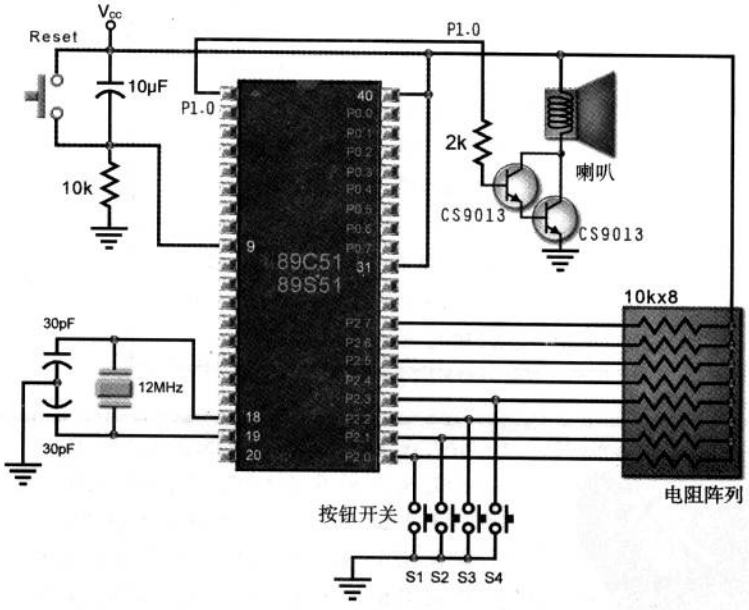


图9-10 快乐点唱机电路图

在如图9-10所示的电路中可以提供四首歌的演奏，按S1将演奏第一首歌，按S2将演奏第二首歌，按S3将演奏第三首歌，按S4将演奏第四首歌。其中第一首歌为“生日快乐歌”，第二首歌是“我是只小小鸟”，第三首歌是“家”，第四首歌是“望春风”。

流程图与程序设计

首先将简谱转化成音谱与节拍，第一首为“生日快乐歌”，其音谱与节拍如下所示。

```
unsigned char song1[]={ 1, 1, 2, 1,  4, 3,  1, 1, 2, 1,  5, 4,
                        1, 1, 8, 6,  4, 3, 9,  7, 7, 6, 4,  5, 4,  15};
unsigned char beat1[]={ 4, 4, 8, 8,  8, 16,  4, 4, 8, 8,  8, 16,
                        4, 4, 8, 8,  8, 8, 8,  4, 4, 8, 8,  8, 16,};
```

第二首为“我是只小小鸟”，其音谱与节拍如下所示。

```
unsigned char song2[]=  {  1, 1, 1,  3, 2, 1,  3, 3, 3,  5, 4, 3,  5, 4, 3,  2,
                           2, 1, 0,  1, 2, 3,  4, 3, 2,  3, 4, 5,  5, 4, 3, 2,  1,  15};
unsigned char beat2[]=  {  4, 4, 4,  6, 2, 4,  4, 4, 4,  6, 2, 4,  4, 4, 4,  12,
                           8, 2, 2,  4, 4, 4,  8, 2, 2  4, 4, 4,  2, 2, 4, 4,  12 };
```

第三首为“家”，简谱与其音谱与节拍如下所示。

C调 3/4　　家

| 1' 7 6 5 5 | 6 1' 5 - | 6 5 3 2 5 | 3 - - 0 |
我 家 门前 有小河，后面 有 山 坡。

| 1' 7 6 5 5 | 6 1' 5 - | 6 5 3 2 5 | 1 - - 0 |
山 坡 上面 野花多，野花红 似 火。

| 2 3 2 5 - | 6 5 6 1' - | 2' 1' 7 6 2' | 5 - - 0 |
小河 里，有白 鹅， 鹅 儿 戏 绿 波。

| 6 1' 5 6 | 3 5 6 5 3 | 2 3 5 3 2 | 1 - - 0 |
戏弄余波，鹅儿 快乐，昂首唱 轻 歌。

```
unsigned char song3[]=  {  8, 7, 6, 5, 5,  6, 8, 5,  6, 5, 3, 2, 5  3, 0,
                           8, 7, 6, 5, 5,  6, 8, 5,  6, 5, 3, 2, 5  1, 0,
                           2, 3, 2, 5,  6, 5, 6, 8,  9, 8, 7, 6, 9  5, 0,
                           6, 8, 5, 6,  3, 5, 6, 5, 3,  2, 3, 5, 3, 2,  1, 0,  15};
unsigned char beat3[]=  {  8, 4, 4, 8, 8,  8, 8, 16,  8, 4, 4, 8, 8,  24, 8,
                           8, 4, 4, 8, 8,  8, 8, 16,  8, 4, 4, 8, 8,  24, 8,
                           8, 4, 4, 16,  8, 4, 4, 16,  8, 4, 4, 8, 8,  24, 8,
                           8, 8, 8, 8,  8, 4, 4, 8, 8,  8, 4, 4, 8, 8,  24, 8, };
```

第四首为“望春风”，简谱与其音谱与节拍如下所示。

C调 4/4 望春风

| 2· 2 3 5 | 6 5 6 7 - | 2'· 7 7 6 5 | 6 - - - |

孤夜无伴守灯下 春风对面吹 十七八岁未出嫁 看着少年家

| 7· 2' 2' 7 2' | 5· 6 6 - | 2· 7 7 6 5 | 5 - - - |

果然标致面肉白 谁家人子弟 想要问伊惊呆势 心内弹琵琶

| 6· 6 7 6 5 | 3 2 3 5 - | 3· 5 6 7 | 2' - - - |

想要郎君做夫婿 意爱在心内 等待何时君来采 青春花当开

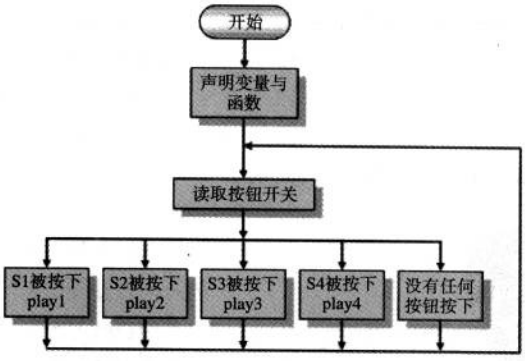
| 2'· 2' 3' 2' 7 | 7 6 5 3 - | 2· 7 7 6 5 | 5 - - - |

听见外面有人来 开门甲看觅 月娘笑我憨大呆 乎风骗不知

```
unsigned char song4[]=  {  2, 2, 3, 5,  6, 5, 6, 7,  9, 7, 7, 6, 5,  6,
                           7, 9, 9, 7, 9,  5, 6, 6,  2, 7, 7, 6, 5,  5,
                           6, 6, 7, 6, 5,  3, 2, 3, 5,  3, 5, 6, 7,  9,
                           9, 9, 10, 9, 7,  7, 6, 5, 3,  2, 7, 7, 6, 5,  5,  15};
unsigned char beat4[]=  {  12, 4, 8, 8,  8, 4, 4, 16  12, 4, 4, 4, 8,  32,
                           12, 4, 8, 4, 4,  12, 4, 16,  12, 4, 8, 4, 4,  32,
                           12, 4, 8, 4, 4,  8, 4, 4, 16,  12, 8, 8, 8,  32,
                           12, 4, 8, 4, 4,  8, 4, 4, 16,  12, 4, 8, 4, 4,  32,  };
```

歌曲内容（song.h）

根据功能需求与电路结构得知，根据按钮开关的状况判断执行哪一首歌的演奏，也就是读取哪个数组。




```
/* ch9-4.c_ 快乐点唱机实验 */
#pragma LA
/* 使用大型(Large)内存模式，将数据放入XDATA区 */
#include <reg51.h>
sbit speaker = P1^0;                    // 声明输出端
```

```
unsigned char  keys,i;                          // 声明按钮及播放谱变量
unsigned char  tone_H, tone_L;                  // 声明定时量变量
void  beat_125(unsigned char);                  // 声明节拍函数
unsigned char code song1[]={     1, 1, 2, 1,  4, 3,  1, 1, 2, 1,  5, 4,
                                 1, 1, 8, 6,  4, 3, 9,  7, 7, 6, 4,  5, 4,  15 };
unsigned char code beat1[]={     4, 4, 8, 8,  8, 16,  4, 4, 8, 8,  8, 16,
                                 4, 4, 8, 8,  8, 8, 8,  4, 4, 8, 8,  8, 16 };
unsigned char song2[]={ 1, 1, 1,  3, 2, 1,  3, 3, 3,  5, 4, 3,  5, 4, 3,  2,
                                 2, 1, 0,  1, 2, 3,  4, 3, 2,  3, 4, 5,  5, 4, 3, 2,  1, 15};
unsigned char beat2[]={  4, 4, 4,  6, 2, 4,  4, 4, 4,  6, 2, 4,  4, 4, 4,  12 ,
                                 8, 2, 2,  4, 4, 4,  8, 2, 2,  4, 4, 4,  2, 2, 4, 4,  12 };
unsigned char song3[]={ 8, 7, 6, 5, 5,  6, 8, 5,  6, 5, 3, 2, 5,  3, 0,
                                 8, 7, 6, 5, 5,  6, 8, 5,  6, 5, 3, 2, 5,  1, 0,
                                 2, 3, 2, 5,  6, 5, 6, 8,  9, 8, 7, 6, 9,  5, 0,
                                 6, 8, 5, 6,  3, 5, 6, 5, 3,  2, 3, 5, 3, 2,  1, 0,  15 };
unsigned char beat3[]={  8, 4, 4, 8, 8,  8, 8, 16,  8, 4, 4, 8, 8,  24, 8,
                                 8, 4, 4, 8, 8,  8, 8, 16,  8, 4, 4, 8, 8,  24, 8,
                                 8, 4, 4, 16,  8, 4, 4, 16,  8, 4, 4, 8, 8,  24, 8,
                                 8, 8, 8, 8,  8, 4, 4, 8, 8,  8, 4, 4, 8, 8,  24, 8 };
unsigned char song4[]={ 2, 2, 3, 5,  6, 5, 6, 7,  9, 7, 7, 6, 5,  6,
                                 7, 9, 9, 7, 9,  5, 6, 6,  2, 7, 7, 6, 5,  5,
                                 6, 6, 7, 6, 5,  3, 2, 3, 5,  3, 5, 6, 7,  9,
                                 9, 9, 10, 9, 7,  7, 6, 5, 3,  2, 7, 7, 6, 5,  5,  15};
unsigned char beat4[]={  12, 4, 8, 8,  8, 4, 4, 16,  12, 4, 4, 4, 8,  32,
                                 12, 4, 8, 4, 4,  12, 4, 16,  12, 4, 8, 4, 4,  32,
                                 12, 4, 8, 4, 4,  8, 4, 4, 16,  12, 8, 8, 8,  32,
                                 12, 4, 8, 4, 4,  8, 4, 4, 16,  12, 4, 8, 4, 4,  32  };
unsigned char tone[]={   1012, 903, 852, 759, 716, 638,
                                 568, 506, 478, 426, 379 };//音阶定义
void play1(void);                               // 声明play1函数
void play2(void);                               // 声明play2函数
void play3(void);                               // 声明play3函数
void play4(void);                               // 声明play4函数
void beat_125(unsigned char);                   // 声明节拍函数
//=================================================
main()
{ speaker=0;                                    // 喇叭初始值
  EA=1;                                         // 开启中断总开关
```

```
    ET0=1;                                  // 开启Timer 0中断开关
    P2=0xff;                                // 将Port2规划为输入口
    while(1)                                // while循环
    {     keys=~P2;                         // 读取按钮
          switch(keys)                      // 判断
          {   case 0x01: play1(); break;    // 按下S1，播放第一首歌
              case 0x02: play2(); break;    // 按下S2，播放第二首歌
              case 0x04: play3(); break;    // 按下S3，播放第三首歌
              case 0x08: play4(); break;    // 按下S4，播放第四首歌
          }
    }                                       // while循环结束
}                                           // 主程序结束
//==第一首歌==================================
void play1(void)
{ while(song1[i]!=15)                       // while循环开始
  {     tone_H=(65536-tone[song1[i]])/256;      // 读取音阶计数量的高八位
        tone_L=(65536-tone[song1[i]])%256;      // 读取音阶计数量的低八位
        TH0=tone_H;                         // 填入音阶计数量的高八位
        TL0=tone_L;                         // 填入音阶计数量的低八位
        TR0=1;                              // 启动Timer 0
        beat_125(beat1[i]);                 // 指定节拍
        i++;                                // 下一个音
  }                                         // 结束播放
}
//==第二首歌==================================
void play2(void)
{ while(song2[i]!=15)                       // while循环开始
  {     tone_H=(65536-tone[song2[i]])/256;      // 读取音阶计数量的高八位
        tone_L=(65536-tone[song2[i]])%256;      // 读取音阶计数量的低八位
        TH0=tone_H;                         // 填入音阶计数量的高八位
        TL0=tone_L;                         // 填入音阶计数量的低八位
        TR0=1;                              // 启动Timer 0
        beat_125(beat2[i]);                 // 指定节拍
        i++;                                // 下一个音
  }
}                                           // 结束播放
//==第三首歌==================================
void play3(void)
```

```
{ while(song3[i]!=15)                          // while循环开始
  {     tone_H=(65536-tone[song3[i]])/256;     // 读取音阶计数量的高八位
        tone_L=(65536-tone[song3[i]])%256;     // 读取音阶计数量的低八位
        TH0=tone_H;                            // 填入音阶计数量的高八位
        TL0=tone_L;                            // 填入音阶计数量的低八位
        TR0=1;                                 // 启动Timer 0
        beat_125(beat3[i]);                    // 指定节拍
        i++;                                   // 下一个音
  }
}                                              // 结束播放
//==第四首歌==================================
void play4(void)
{ while(song1[i]!=15)                          // while循环开始
  {     tone_H=(65536-tone[song4[i]])/256;     // 读取音阶计数量的高八位
        tone_L=(65536-tone[song4[i]])%256;     // 读取音阶计数量的低八位
        TH0=tone_H;                            // 填入音阶计数量的高八位
        TL0=tone_L;                            // 填入音阶计数量的低八位
        TR0=1;                                 // 启动Timer 0
        beat_125(beat4[i]);                    // 指定节拍
        i++;                                   // 下一个音
  }
}                                              // 结束播放
//====Timer 0中断子程序==================
void tone_timer(void)interrupt 1               // Timer 0中断子程序开始
{ TH0=tone_H;                                  // 填入定时量的高八位
  TL0=tone_L;                                  // 填入定时量的低八位
  speaker=~speaker;                            // 喇叭反相输出
}                                              // 结束中断子程序
//====节拍函数==================
void beat_125(unsigned char x)                 // 节拍函数开始
{ unsigned char i,j,k;                         // 声明变量
  for(i=0;i<x;i++)                             // i循环
      for(j=0;j<125;j++)                       // j循环
          for(k=0;k<120;k++);                  // k循环
}                                              // 结束节拍函数
```

快乐点唱机实验一（ch9-4.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图9-35连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

思考一下

1. 本实验的程序很长，有点像是土法炼钢，可否利用“指标”的方式，将4个类似的play函数，变成单一个函数？

2. 如果要以两个定时器来完成本实验的功能，程序应如何编写？

9.5.5 快乐点唱机实例演练二

实验要点

本实验与9.5.4节相同，而采用程序与数据分开的方式，让点唱机更实用。

流程图与程序设计

首先将音谱与节拍存为song.h文件，如下所示。

```
unsigned char song1[]={ 1, 1, 2, 1,  4, 3,  1, 1, 2, 1,  5, 4,
                        1, 1, 8, 6,  4, 3, 9,  7, 7, 6, 4,  5, 4,  15 };
unsigned char beat1[]={ 4, 4, 8, 8,  8, 16,  4, 4, 8, 8,  8, 16,
                        4, 4, 8, 8,  8, 8, 8,  4, 4, 8, 8,  8, 16 };
unsigned char song2[]={ 1, 1, 1,  3, 2, 1,  3, 3, 3,  5, 4, 3,  5, 4, 3,  2,
                        2, 1, 0,  1, 2, 3,  4, 3, 2,  3, 4, 5,  5, 4, 3, 2,  1, 15};
unsigned char beat2[]={ 4, 4, 4,  6, 2, 4,  4, 4, 4,  6, 2, 4,  4, 4, 4,  12 ,
                        8, 2, 2,  4, 4, 4,  8, 2, 2,  4, 4, 4,  2, 2, 4, 4,  12 };
unsigned char song3[]={ 8, 7, 6, 5, 5,  6, 8, 5,  6, 5, 3, 2, 5,  3, 0,
                        8, 7, 6, 5, 5,  6, 8, 5,  6, 5, 3, 2, 5,  1, 0,
                        2, 3, 2, 5,  6, 5, 6, 8,  9, 8, 7, 6, 9,  5, 0,
                        6, 8, 5, 6,  3, 5, 6, 5, 3,  2, 3, 5, 3, 2,  1, 0,  15 };
```

```
unsigned char beat3[]={  8, 4, 4, 8, 8,  8, 8, 16,  8, 4, 4, 8, 8,  24, 8,
                         8, 4, 4, 8, 8,  8, 8, 16,  8, 4, 4, 8, 8,  24, 8,
                         8, 4, 4, 16,  8, 4, 4, 16,  8, 4, 4, 8, 8,  24, 8,
                         8, 8, 8, 8,  8, 4, 4, 8, 8,  8, 4, 4, 8, 8,  24, 8 };
unsigned char song4[]={ 2, 2, 3, 5,  6, 5, 6, 7,  9, 7, 7, 6, 5,  6,
                         7, 9, 9, 7, 9,  5, 6, 6,  2, 7, 7, 6, 5,  5,
                         6, 6, 7, 6, 5,  3, 2, 3, 5,  3, 5, 6, 7,  9,
                         9, 9, 10, 9, 7,  7, 6, 5, 3,  2, 7, 7, 6, 5,  5,  15};
unsigned char beat4[]={  12, 4, 8, 8,  8, 4, 4, 16,  12, 4, 4, 4, 8,  32,
                         12, 4, 8, 4, 4,  12, 4, 16,  12, 4, 8, 4, 4,  32,
                         12, 4, 8, 4, 4,  8, 4, 4, 16,  12, 8, 8, 8,  32,
                         12, 4, 8, 4, 4,  8, 4, 4, 16,  12, 4, 8, 4, 4,  32  };
```

改写程序如下所示。

```
/* ch9-5.c_ 快乐点唱机实验 */
#pragma LA
#include  <reg51.h>
#include "song.h"
sbit  speaker = P1^0;                                  // 声明输出端
unsigned char  keys,i;                                 // 声明按钮及播放谱变量
unsigned char  tone_H, tone_L;                         // 声明定时量变量
void  beat_125(unsigned char);                         // 声明节拍函数
unsigned char tone[]={   1012, 903, 852, 759, 716, 638,
                         568, 506, 478, 426, 379 };          // 音阶定义
void play(unsigned char *,unsigned char *);                  // 声明play函数
void beat_125()unsigned char ) ;                             // 声明节拍函数
//================================================
main()
{ speaker=0;                                           // 喇叭初始值
  EA=1;                                                // 开启中断总开关
  ET0=1;                                               // 开启Timer 0中断开关
  P2=0xff;                                             // 将Port2规划为输入口
  while(1)                                             // while循环
  {     i=0;
        keys=~P2;                                      // 读取按钮
        switch(keys)                                   // 判断
        {    case 0x01: play(song1,beat1); break;      // 按下S1，播放第一首歌
```

```
            case 0x02: play(song2,beat2); break;   // 按下S2，播放第二首歌
            case 0x04: play(song3,beat3); break;   // 按下S3，播放第三首歌
            case 0x08: play(song4,beat4); break;   // 按下S4，播放第四首歌
            }
  }                                                // while循环结束
}                                                  // 主程序结束
//==播放歌==================================
void play(unsigned char* song,unsigned char* beat)
{ while(song1[i]!=15)                              // while循环开始
  {     tone_H=(65536-tone[song[i]])/256;          // 读取音阶计数量的高八位
        tone_L=(65536-tone[song[i]])%256;          // 读取音阶计数量的低八位
        TH0=tone_H;                                // 填入音阶计数量的高八位
        TL0=tone_L;                                // 填入音阶计数量的低八位
        TR0=1;                                     // 启动Timer 0
        beat_125(beat[i]);                         // 指定节拍
        i++;                                       // 下一个音
  }                                                // 结束播放
}
//====Timer 0中断子程序==================
void tone_timer(void)interrupt 1                   // Timer 0中断子程序开始
{ TH0=tone_H;                                      // 填入定时量的高八位
  TL0=tone_L;                                      // 填入定时量的低八位
  speaker=~speaker;                                // 喇叭反相输出
}                                                  // 结束中断子程序
//====节拍函数==================
void beat_125(unsigned char x)                     // 节拍函数开始
{ unsigned char i,j,k;                             // 声明变量
  for(i=0;i<x;i++)                                 // i循环
      for(j=0;j<125;j++)                           // j循环
          for(k=0;k<120;k++);                      // k循环
}                                                  // 结束节拍函数
```

快乐点唱机实验二（ch9-5.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图9-35连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接通电，看看是否正常？

5. 编写实验报告。

9.6 实时练习



在本章中探讨了声音的原理与8051产生声音的方法，基本上包括音频与节拍两部分，除了程序的介绍外，也介绍了如何根据乐谱编写产生音乐的程序。在此请试着回答下列问题，以确认对于此部分的掌握程度。

选择题

（　）1. 若要使用8051演奏音乐，除了音阶外，还要处理哪个项目？
（A）歌曲长度 （B）节拍 （C）高低音 （D）声音大小

（　）2. 在8051里要产生不同的音阶，可采用什么方法？
（A）定时器与外部中断 （B）外部中断与延迟函数
（C）延迟函数与定时器 （D）以上皆可

（　）3. 若要产生1 kHz的声音，则8051必须多久切换一次输出状态？
（A）0.5ms （B）1ms （C）2ms （D）4ms

（　）4. 音频的范围是多少？
（A）2kHz~200kHz （B）200Hz~2MHz
（C）20Hz~2MHz （D）20Hz~200kHz

（　）5. 在8051产生声音的电路里，以何种波形驱动喇叭？
（A）正弦波 （B）脉冲 （C）三角波 （D）直流电

（　）6. 若要以8051的Port 0来驱动蜂鸣器，应如何处理？
（A）直接连接晶体管的基极，而晶体管的集极再连接蜂鸣器
（B）直接连接蜂鸣器
（C）连接晶体管的基极，同时连接一个上拉电阻，而晶体管的集极再连接蜂鸣器
（D）连接一个交连电容，再连接蜂鸣器

（　）7. 高音Do频率是中音Do频率的多少倍？
（A）两倍频 （B）中音= $\sqrt[12]{2}$ × 高音Do （C）高音= $\sqrt[12]{2}$ × 中音Do （D）一半频率

（　）8. Do与Do#的频率关系是什么？
（A）Do=2 Do# （B）Do#= $\sqrt[12]{2}$ × Do （C）Do= $\sqrt[12]{2}$ × Do# （D）Do=2 Do#

（　）9. 在歌谱上的“C 3/4”代表什么？
（A）4小节、每小节3拍 （B）3小节、每小节4拍

（C）总共4小节、目前是第3小节 （D）总共4拍、目前是第3拍

（ ）10. 在12MHz的8051系统里，若要以for语句产生1ms的时间延迟，此循环大约要重复多少次？

（A）10 （B）120 （C）1500 （D）6000



1. 试问声音的频率范围是多少？
2. 若要使用8051的Port 0驱动喇叭，必须注意什么？
3. 试问中音的Do频率为多少？而两个半音之间的频率比率是多少？
4. 试问八音度有多少个半音？而每个八音度之间频率差异多少？
5. 试说明在乐谱左上方所标注的“C 4/4”，代表什么意义？

(C) 恰共4小行，自测题第1页 (D) 机车相，日行[illegible]

(　) 10.在1,2,3,4,5,6,7,8,9这9个数字中，若[illegible]的学生[illegible]时间参加，且需在大约

(A) 10 (B) 30 (C) 1500 (D) 4000

[illegible]

1. 哪两个[illegible]

2. 若要[illegible]

3. 从[illegible]

4. 从[illegible]

5. 共[illegible]

第10章 步进马达的控制

本章内容丰富，主要包括以下两部分。

硬件部分：

步进马达的结构、驱动方式以及如何应用8051来控制步进马达等。

程序与实作部分：

步进马达的1相驱动、2相驱动与1-2相驱动程序的应用。

步进马达的角度控制、速度控制等。

10.1 认识步进马达



步进马达（stepping motor）是一种用脉冲控制的转动设备，由于是用脉冲驱动，很适合用数字或微型计算机来控制，所以通常把它当成是一种数字设备。

10.1.1 步进马达的结构

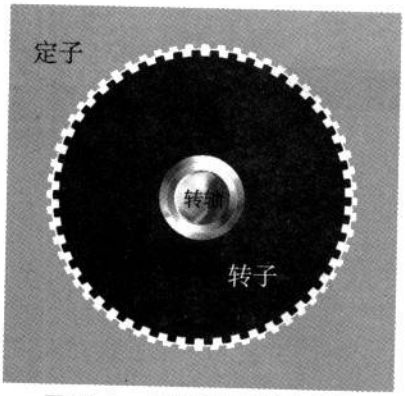


图10-1 步进马达的基本结构

步进马达与一般马达结构类似，除了托架、外壳之外，就是转子与定子，比较特殊的是其转子与定子上有许多细小的齿。其转子是永久磁铁，线圈绕在定子上。根据线圈的配置，步进马达可分为2相、4相、5相等，如图10-2所示。比较常用的是2相的步进马达，其中包括两组具有中间抽头的线圈，A、com1与 $\overline{A}$ 为一组，B、com2与 $\overline{B}$ 为另一组。2相6线式步进马达，其连接线就是A、com1、 $\overline{A}$ 、B、com2与 $\overline{B}$；而2相5线式步进马达是将其中的com1与com2连接。另外，4相步进马达是由4组线圈所构成的，5相步进马达是由5组线圈所构成的。

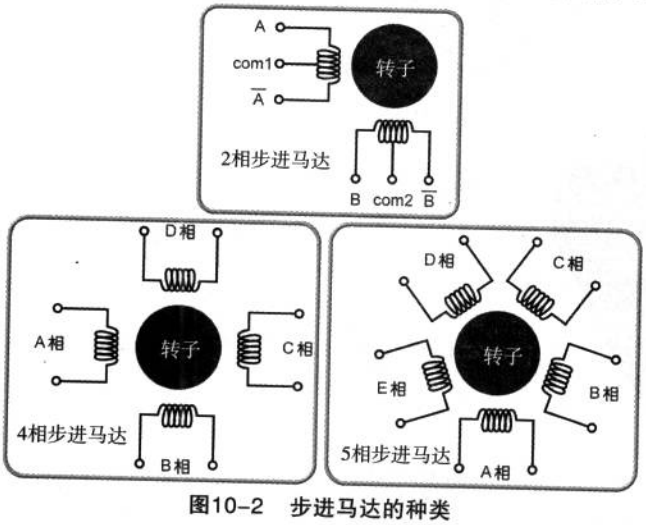


图10-2 步进马达的种类

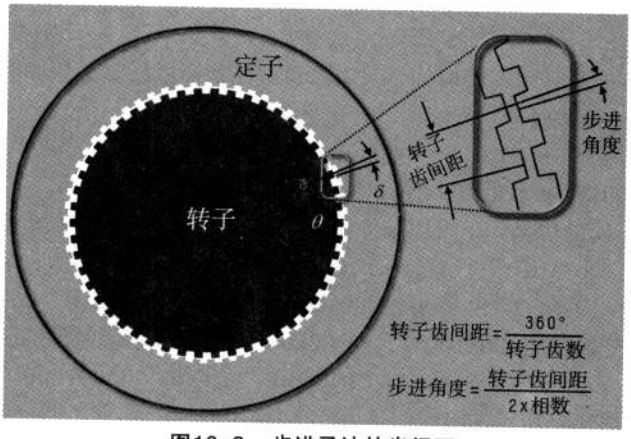


图10-3 步进马达的齿间距

顾名思义，步进马达就是一步步走的马达，而其转子与定子的齿，决定了每步的间距，若转子上有N个齿，则其齿间距θ为

$$\theta=\text{转子齿间距}=\frac{360^\circ}{N}$$

而步进角度δ为

$$\delta=\frac{\text{转子齿间距}}{2\times\text{相数}}=\frac{\theta}{2P}$$

以常用的2相式50齿步进马达为例，

$\theta=360^\circ/50=7.2^\circ$

$\delta=7.2^\circ/(2\times2)=1.8^\circ$

另外还有一种比较简便的说法，就是以步数来表示，以200步的步进马达为例，200步为一圈（360°），则每步1.8°。

10.1.2 步进马达的动作

简单讲，步进马达的动作是靠定子线圈激磁后将邻近转子上相异磁极吸引过来实现的。因此，线圈排列的顺序以及激磁信号的顺序就很重要！以2相式步进马达为例，其驱动信号有1相驱动、2相驱动与1-2相驱动三种，如图10-4所示。

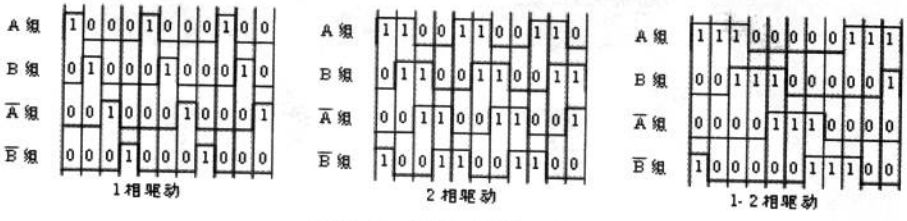


图10-4 步进马达的驱动方式

1相驱动

1相驱动的方式是任何一个时间，只有一组线圈被激磁，其他线圈在休息，因此产生的力矩较小。但这种激磁方式最简单，其信号依次为：

1000 ⟶ 0100 ⟶ 0010 ⟶ 0001 ⟶ 1000…（正转）

1000 ⟶ 0001 ⟶ 0010 ⟶ 0100 ⟶ 1000…（反转）

总共有4种不同的信号，呈现周期性的变化。在8051中，若要产生左移信号，可先输出“00000001”，即“0x01”。经过一小段的时间延迟，让步进马达有足够的时间建立磁场及转动，再以“P1<<=1”指令将“00000001”左移，使它变为“00000010”。若要产生右移信号，可先输出“00001000”，即“0x08”。经过一小段时间的延迟，让步进马达有足够的时间建立磁场及转动，再以“P1>>=1”指令将“00001000”左移，使它变为“00000100”，如图10-5所示的流程图是由Port 1输出的。

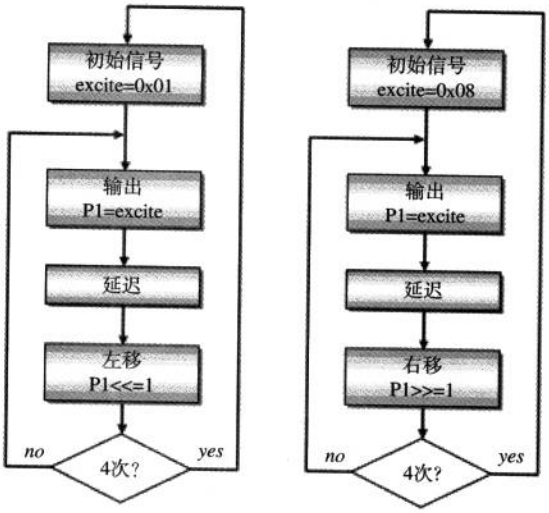
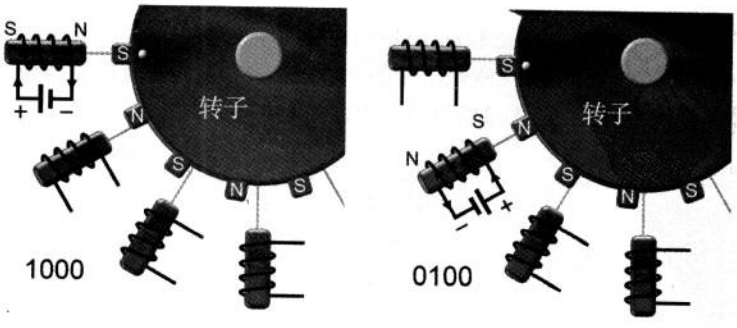


图10-5 1相驱动信号的产生（左正转、右反转）

这4个驱动信号为“0x01、0x02、0x04、0x08”，若将这4个信号依次加入步进马达，其反应如图10-6所示。



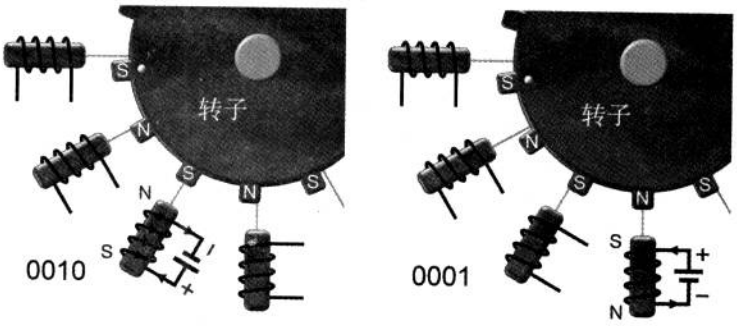


图10-6 1相驱动步进马达的动作

2相驱动

2相驱动的方式是在任何一个时间内都有两组线圈同时被激磁，因此，其所产生的力矩比1相驱动大。尽管如此，这种激磁方式也很简单，其信号依次为：

1100 ⟶ 0110 ⟶ 0011 ⟶ 1001 ⟶ 1100…（正转）

1100 ⟶ 1001 ⟶ 0011 ⟶ 0110 ⟶ 1100…（反转）

总共有4种不同的信号呈现周期性的变化，即“0x0c、0x06、0x03、0x09”。我们可把这组信号存入数组，再依次从数组读出，经过一小段时间的延迟，让步进马达有足够的时间建立磁场及转动。若要反方向转动，则从数组反序读出，经过一小段的时间延迟，让步进马达有足够的时间建立磁场及转动，如图10-7所示的流程图是由Port 1输出的。

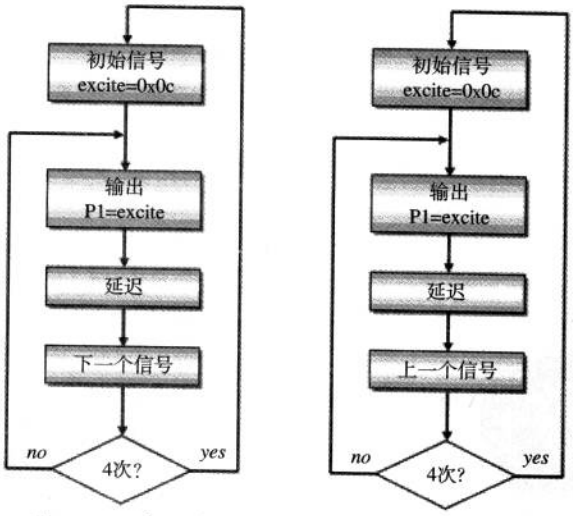


图10-7 2相驱动信号的产生（左正转、右反转）

1-2相驱动

1-2相驱动的方式又称为“半步驱动”，每个驱动信号只驱动半步。而其驱动信号依次为：

1001 ⟶ 1000 ⟶ 1100 ⟶ 0100 ⟶ 0110 ⟶ 0010 ⟶ 0011 ⟶ 0001（正转）

1001 ⟶ 0001 ⟶ 0011 ⟶ 0010 ⟶ 0110 ⟶ 0100 ⟶ 1100 ⟶ 1000（反转）

总共有8种不同的信号，呈现周期性的变化，仔细观察可发现其中的信号是将1相驱动信号与2相驱动信号混合而成。为了方便起见，在此依次以16进制方式，列出这4个信号，即“0x09、0x08、0x0c、0x04、0x06、0x02、0x03、0x01”，其中蓝色字为2相驱动信号、黑色字为1相驱动信号。

在8051里，可先这8个信号放入数组，再依次读出、输出，其流程图如图10-8所示。当然，两个信号输出的中间，还是需要经过一小段的时间延迟，让步进马达有足够的时间建立磁场及转动。

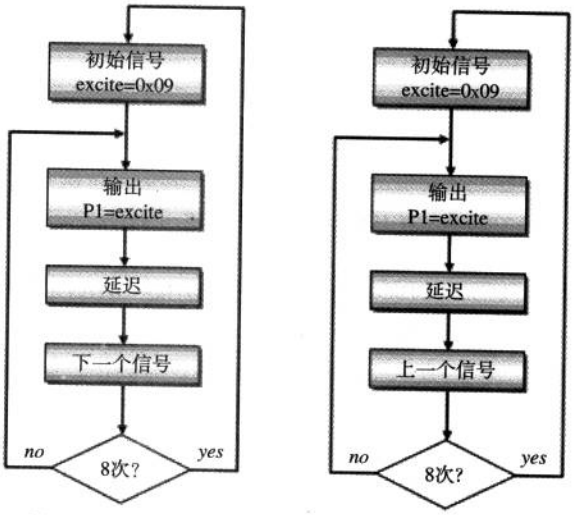


图10-8 1-2相驱动信号的产生（左正转、右反转）

10.1.3 步进马达的定位

当我们打开个人计算机时，计算机上的软驱会动一下、连接该计算机的外围设备也会有所反应。不管是动一下、闪一下或反应一下，都是RESET的动作，让所有设备归为一定的状态。若说步进马达是一种数字输出设备，则这种设备在使用之前必须归零或定位，才能精确地使用这个步进马达。在图10-6中，将“1000、0100、0010、0001”信号加入步进马达时，该步进马达将逆时针转动4步，若每步为1.8°，则总共转了7.2°。同样的驱动信号，如果一开始步进马达的转子位置不对，则可能发生以下两种非预期状态。

先顺时针转再逆时针转

如图10-9所示，送入“1000”信号时，步进马达顺时针旋转1.8°，如图10-10所示。

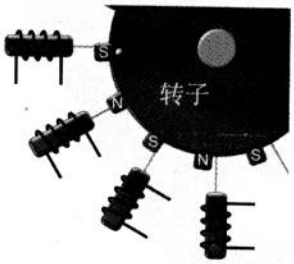


图10-9 原始转子位置

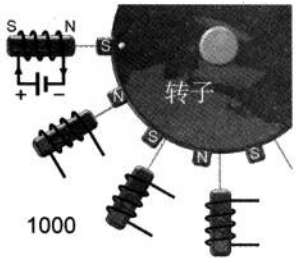


图10-10 顺时针旋转1.8°

到第二组信号（“0100”、“0010”、“0001”）才开始逆时针旋转，如图10-11所示。

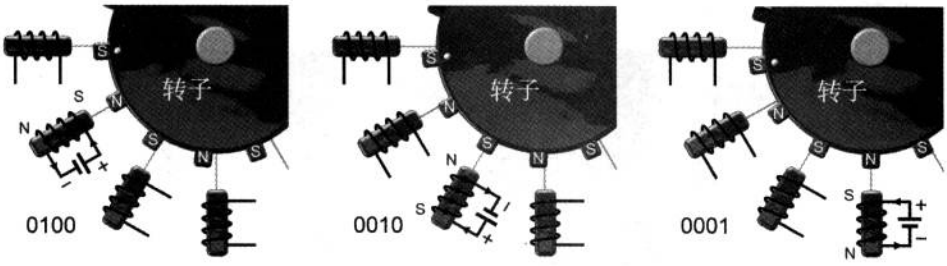


图10-11 逆时针旋转5.4°

如此一来，总共逆时针旋转3.6°，而非预期的逆时针旋转7.2°。

先抖动，再顺时针转，然后逆时针转

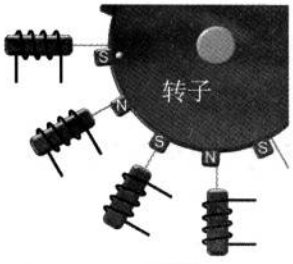


图10-12 原始转子位置

如图10-12所示，送入“1000”信号时，步进马达抖动，如图10-13所示。

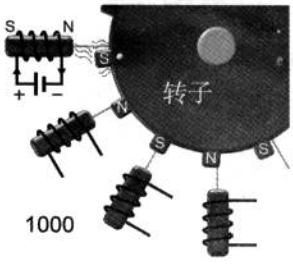


图10-13 吸不过来

到第二组信号（“0100”）时，步进马达顺时针旋转1.8°，如图10-14所示。

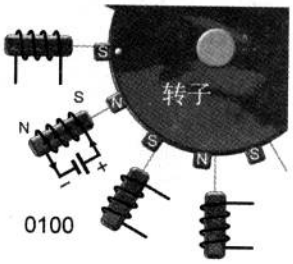


图10-14　顺时针旋转1.8°

到第三组信号（“0010”、“0001”）时，才开始逆时针旋转，如图10-15所示。

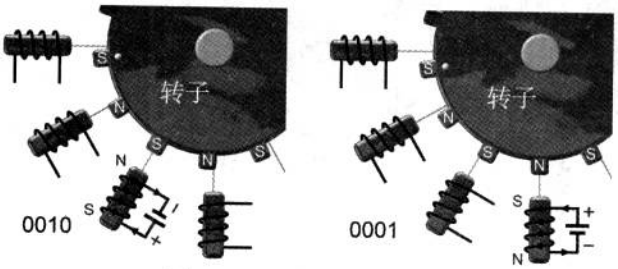


图10-15　逆时针旋转3.6°

如此一来，总共逆时针旋转1.8°，而非预期的逆时针旋转7.2°。

为了防止出现上述非预期状况，最简单的方法是在开始运作之前，先送出一组信号，换言之，若是1相或2相驱动，则依次送出4个驱动信号；若是1-2相驱动，则依次送出8个驱动信号，即可设置正确的步进马达位置，通常把该过程称为定位或归零。

10.2 步进马达驱动电路



8051的输出电流很难驱动步进马达，必须另外设置驱动电路才行，在此将介绍几种常用的驱动电路。

10.2.1 小型步进马达的驱动电路

对于电流小于0.5A的步进马达，可以采用ULN2003之类的驱动IC，它是一种“小而美”的驱动设备，可不要看它只是个一般包装的IC，它所提供的输出电路（吸入）可以达到0.5A！图10-16所示为2003系列驱动IC的引脚图。

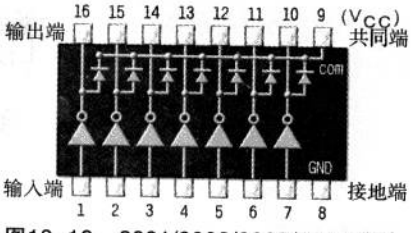


图10-16　2001/2002/2003/2004系列驱动IC的引脚图

一片2003系列IC包含7个开集极式输出的反相器，而在每个输出端都有一个连接到共同端（V_{CC}）的二极管，作为放电保护电路，每组反相器的内部电路如图10-17所示。

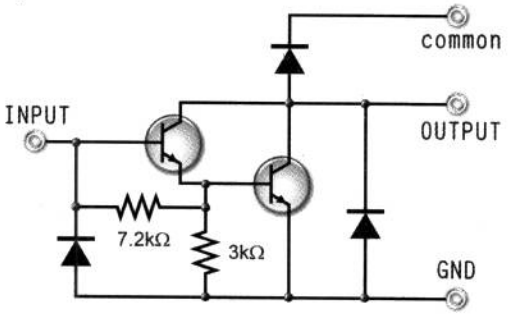


图10-17 内部电路

如图1-17所示，基本上这是一个附有保护二极管的达林顿电路，它可提供最高0.5A的电流。对于步进马达而言，可能会有瞬间的大电流，但每个驱动电路所提供的工作周期都不高，驱动IC应该不会过热。如图10-18所示，驱动信号由8051的P1.0~P1.3连接到2003系列IC的4个反相器输入端，而其输出端连接到步进马达。2003系列IC的common端与步进马达的com1、com2连接到+5V（或+12V）的电源上，而2003系列IC的第8脚接地。

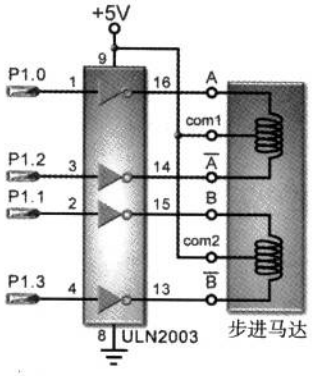


图10-18 2003系列IC驱动电路

10.2.2 达林顿电晶体驱动电路

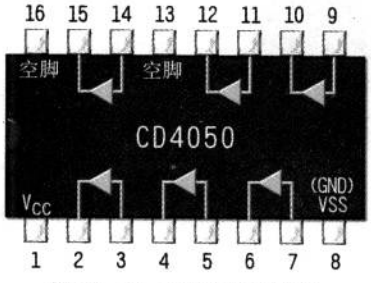


图10-19 CD4050引脚图

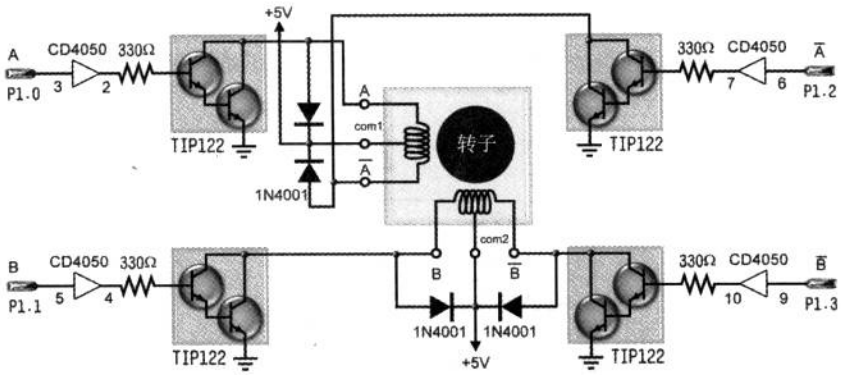


图10-20 达林顿晶体管驱动电路

通常2003系列IC只能驱动较小型的步进马达，对于稍微大一点的步进马达就无能为力了！如果不放心那片小小的IC，或要驱动稍大的步进马达，则可利用中功率包装（TO-220）的达林顿晶体管，如TIP122等。这种晶体管可瞬间放大电流达到1~3A，足以驱动大多数的步进马达。当然，不管使用哪个晶体管或达林顿晶体管，还是需要足够的输入电流（I_B）才能输出较大的驱动电流。而在高电平时，8051的任何一个输出口的输出的电流都非常微小（Port 0还需要外接上拉电阻），最好能先经过一个CMOS的寄存器（以CD4050为例，如图10-19所示），才能确保提供足够的基极电流（I_B）。当然在每相的驱动电路里，还是需要一个基极电阻，以抑制过大的基极电流，而在输出端，也要连接一个1N4001，提供电感器（步进马达的线圈）的放电路径，如图10-20所示。若驱动电流仍然不够，则可使用2N3053与2N3055搭接成达林顿电路，以取代图10-20中的TIP122达林顿晶体管，即可提供更多的驱动电流。

10.2.3 FT5754驱动电路

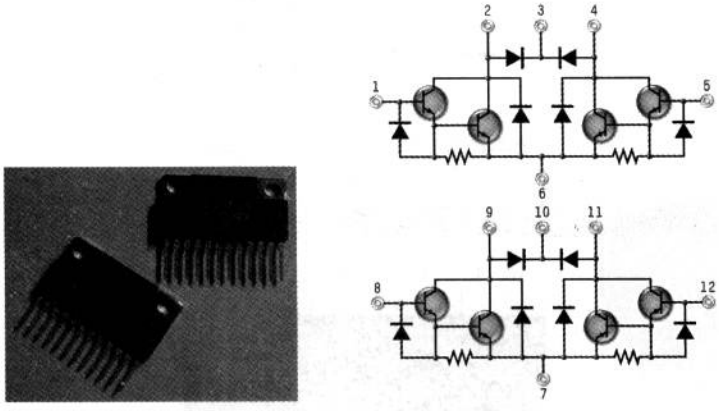


图10-21 FT5754电路

如果觉得使用4个TP122达林顿晶体管很麻烦而且太占地方的话，则FT5754是不错的选择！FT5754是步进马达专用的步进马达驱动IC，它是一片12只引脚的功率IC，如图10-21所示为其外观与其内部电路。

FT5754内部电路包括4组相同的达林顿模块，电路结构与2003系列IC的内部电路有点相似，不过，FT5754所能提供的电流更大！所以，我们可以用它来驱动较大的步进马达。尽管如此，若要用8051（或其他微控制器）的输出端口来驱动这片IC，还是要先经过一个寄存器，图10-22所示为以CD4050为寄存器的FT5754步进马达驱动电路。

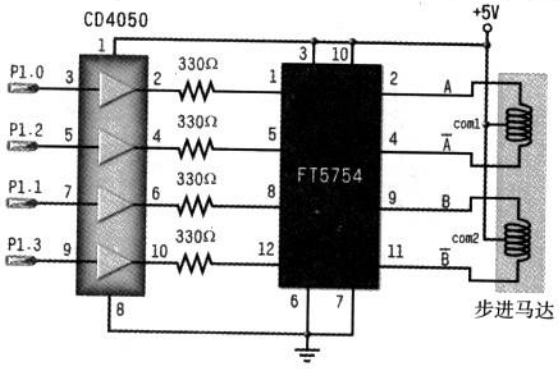


图10-22 FT5754步进马达驱动电路

从表面上来看，图10-22所示的步进马达驱动电路与图10-20所示的TP122步进马达驱动电路差不多，但在实际的电路板上，FT5754步进马达驱动电路简单多了！

10.3 实例演练

在本单元中提供了4个实例，以展示步进马达的控制方法。

10.3.1 以延迟函数产生驱动信号实例演练

● 实验要点

如图10-23所示，在此使用小型的步进马达，所以直接采用ULN2003驱动电路，若使用较大的步进马达，则改用达林顿晶体管驱动电路，或FT5754驱动电路等。本单元将利用延迟函数来控制旋转的速度，采用1相激磁方式。当程序开始时先定位，然后不断地正转或反转。

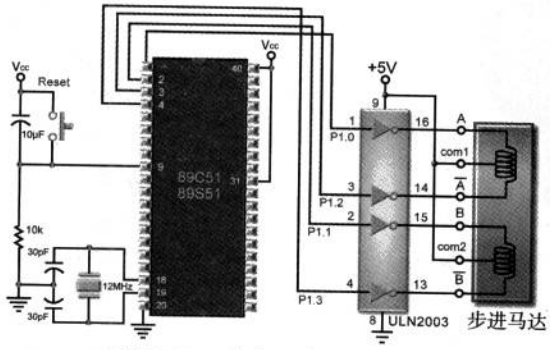
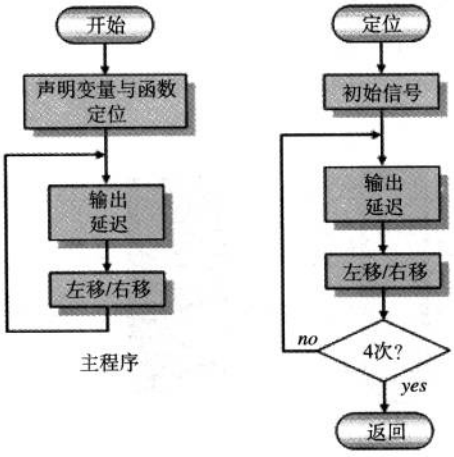


图10-23 步进马达控制电路图

流程图与程序设计



根据功能需求与电路结构可知，其控制程序与单灯左移（1相激磁）类似，而其中的延迟函数将产生0.05s倍数的延迟，其倍数则在程序开始的“unsigned char times=10;”中声明。同样地，激磁方式也可以在程序开始的“unsigned char excite=1;”中声明。另外，在此由P1.0到P1.3连接到步进马达驱动电路，若要改由其他输出口输出，则可以在“#define OUTPUT=P1”中定义。

```
/* ch10_1.c – 1相驱动实验 */
//利用delay50mDELAY子程序(0.05s × times)，产生驱动信号
//由P1输出        速度为 1/(0.05 × timers)步/s

#include <reg51.h>                          // 包含reg51.h文件
#define OUTPUT        P1                    // 定义输出口为P0
unsigned char times=10;                     // 声明延迟时间变量(×50ms)
void step_rst(void);                        // 声明定位函数
void delay50ms(int);                        // 声明延迟函数
//=====主程序=========================================
main()                                      // 主程序开始
{ OUTPUT=0;                                 // 关闭输出
  step_rst();                               // 定位
  while(1)                                  // while循环
  { char i;                                 // 声明变量
    for(i=0;i<4;i++)                        // for 循环
  {     OUTPUT=1<<i;                        // 输出激磁信号(左转)
        delay50ms(times); }                 // 延迟
  }                                         // while循环结束
```

```
}                                                    // 结束主程序
//======定位函数=============================================
void step_rst(void)                                  // 定位函数开始
{ char i;                                            // 声明变量
  for(i=0;i<4;i++)                                   // 输出4个信号
  {  OUTPUT=1<<i;                                    // 输出激磁信号(左转)
     delay50ms(20);                                  // 延迟1s
  }                                                  // 结束
}
//=======延迟函数=========================================
void delay50ms(int x)                                // 延迟函数开始
{ int i,j;                                           // 声明变量
  for(i=0;i<x;i++)                                   // i循环
     for(j=0;j<6000;j++); }                          // j循环
```

1相驱动实验（ch10-1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图10-23连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 当times参数为10时，每步之间的时间差距约0.5s，则步进马达转一圈（200步）应该是100s，请实际量测，看看是否一致？

2. 试着增加times参数，看看是否能使步进马达的速度变慢？再试着减少times参数，看看是否能使步进马达的速度变快？达到多快时，步进马达就不能正常转动？

3. 若要从P2输出到步进马达，要如何修改程序？

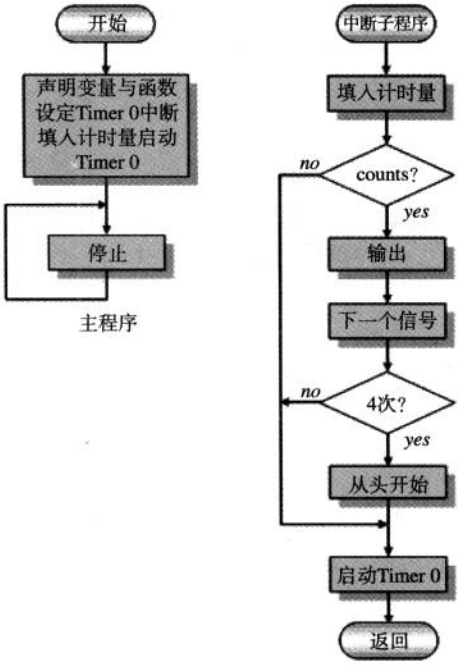
10.3.2 以定时器产生驱动信号实例演练

实验要点

同10.3.1节的电路（如图10-23所示），改用定时器的方式替代延迟函数。

流程图与程序设计

若要以定时器的方式产生时间延迟，则在主程序里设定所有定时器中断的设定，并启动定时器后，就让程序在原地打转，主要的动作是在中断子程序里操作。对于MODE 1而言，每次中断最多只能延迟0.065s，时间太短了！所以，在此只设定为50000，以产生0.05s的中断，再利用times作为计次设备，每中断times次，改变一次输出的信号。如此就能以改变times的次数控制步进马达的速度。另外，把激磁信号放入excite[]数组里，再一个个读出，以驱动步进马达。若excite[]数组里放置1相驱动信号，则可以1相驱动步进马达；若excite[]数组里放置2相驱动信号，则可以2相驱动步进马达。在此将采用2相驱动方式。




```
/* ch10_2.c – 2相驱动实验 */
//利用timer中断子程序(0.05s × times)，产生驱动信号
//由P1输出
//速度为 1/(0.05 × timers)步/s
//=======================================================
#include  <reg51.h>                          // 包含reg51.h文件
#define  OUTPUT   P1                         // 定义输出口为P0
/* 2相驱动激励信号数组 */
char excite[]={ 0x03, 0x06, 0x0c, 0x09};
unsigned char times=10;                      // 声明重复数(×50ms)
unsigned char counts=10;                     // 声明重复变量(=times)
```

```
T_H=(65536-50000)/256;                    // 声明计时量的高8位
T_L=(65536-50000)%256;                    // 声明计时量的低8位
unsigned char i=0;                        // 声明变量
main()                                    // 主程序
{ OUTPUT=0;                               // 关闭输出
  IE=0x82;                                // 启用Timer 0
  TH0=T_H;                                // 填入计时量的高8位
  TL0=T_L;                                // 填入计时量的低8位
  TR0=1;                                  // 启动Timer 0
  while(1);                               // 停滞
}                                         // 结束主程序
//=========================================================
void timer0(void)interrupt 1              // Timer 0中断子程序开始
{ TH0=T_H;                                // 填入计时量的高8位
  TL0=T_L;                                // 填入计时量的低8位
  if(--counts==0)                         // 判断次数
  { OUTPUT=excite[i];                     // 输出
    i++;                                  // 下一个信号
    if(i==4)i=0;                          // 如果超过4组，则从头开始
    counts=times; }                       // 重填
}                                         // Timer 0中断子程序结束
```

2相驱动实验（ch10-2.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图10-23连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 请修改程序，采用1相激磁方式驱动步进马达？

2. 在本实验里，若将步进马达转速提高（times减少），会不会让步进马达不动？若步进马达不动，其驱动信号的频率（或周期）是多少？

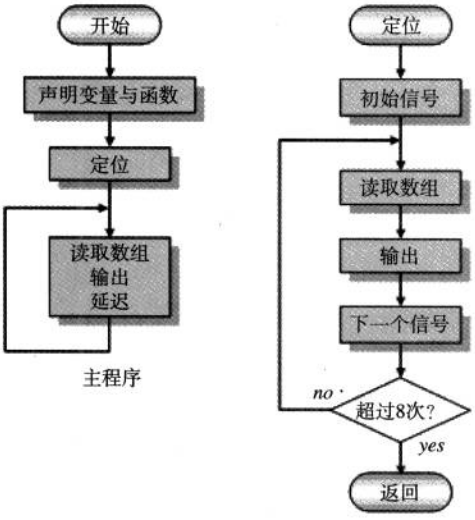
10.3.3 1-2相驱动实例演练

● 实验要点

同10.3.3节的电路（如图10-23所示），但采用1-2相激磁方式输出。

● 流程图与程序设计

1-2相激磁信号是夹杂着1相激磁信号与2相激磁信号，很难以前述的左移或右移方式产生。在此将以数组的方式，将这8个激磁信号放入数组之中，再逐一取出、输出即可。




```
/* ch10_3.c - 1-2相驱动实验 */
#include <reg51.h>                          // 包含reg51.h文件
#define OUTPUT P1                           // 定义输出口为P0
/* 1-2相驱动激励信号数组 */
char excite[]={0x01, 0x03, 0x02, 0x06, 0x04, 0x0c, 0x08, 0x09};
unsigned char times=10;                     // 声明延迟时间变量(×50ms)
void step_rst(void);                        // 声明定位函数
void delay50ms(int);                        // 声明延迟函数
main()                                      // 主程序
{ char i;                                   // 声明变量
  OUTPUT=0;                                 // 关闭输出
  step_rst();                               // 定位
  while(1)                                  // while循环
  { for(i=0;i<8;i++)                        // for循环
    { OUTPUT=excite[i];                     // 输出激磁信号
```

```
        delay50ms(times); }                    // 延迟
  }                                            // while循环结束
}                                              // 结束主程序
//=====================================================
void step_rst(void)                            // 定位函数开始
{ char i;                                      // 声明变量
  for(i=0;i<8;i++)                             // 输出8个信号
  { OUTPUT=excite[i];                          // 输出激磁信号
    delay50ms(20);                             // 延迟1s
  }                                            // 结束
}
//=====================================================
void delay50ms(int x)                          // 延迟函数开始
{ int i,j;                                     // 声明变量
  for(i=0;i<x;i++)                             // i循环
    for(j=0;j<6000;j++); }                     // j循环
```

1-2相驱动实验（ch10-3.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若有错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图10-23连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

思考一下

1. 如何将本实验里的步进马达，由正转变成反转？

2. 在本实验里，步进马达转一圈需要多少时间？

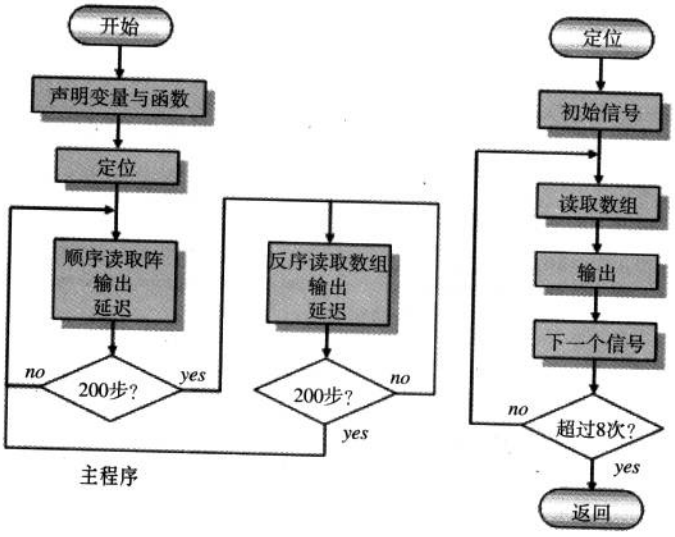
10.3.4 方向控制实例演练

实验要点

同10.3.1节的电路（如图10-23所示），而在此将先让步进马达正转一圈（200步），再反转一圈。

流程图与程序设计

根据功能需求与电路结构可知，最简单的方式还是采用延迟函数的方式，先以左移方式输出200个信号，再以右移方式输出200个信号。顺序读取excite数组的方法，可将所要执行的第几步（即i）除以4的余数，作为读取所要数组中的第几个信号，例如i=0时，i除以4的余数为0，将excite[0]输出；i=1时，i除以4的余数为1，将excite[1]输出…i=150时，i除以4的余数为2，将excite[2]输出；i=151时，i除以4的余数为3，将excite[3]输出…依次类推。我们可利用“OUTPUT=excite[i%4];”指令，即可达到顺序输出的目的。若要反序输出，则可使用“OUTPUT=excite[3-i%4];”指令。若要改变为2相激磁，只要改变excite数组的内容即可，而不必更改程序。




```
/* ch10_4.c - 方向控制实验 */
//由P1输出，正转200步(1圈)，再反转200步(1圈)，
//excite[]为1相驱动，更改内容可改为2相驱动
//速度为 1/(0.05 TIMES)步/s
//=================================================
#include <reg51.h>                          // 包含reg51.h文件
#define OUTPUT P1                           // 定义输出口为P0
unsigned char steps=20;                     // 声明步数变量
unsigned char times=10;                     // 声明延迟时间变量(×50ms)
/* 1相激励信号数组 */
char excite[]={ 0x01, 0x02, 0x04, 0x08};
/* 2相驱动数组内容={ 0x03, 0x06, 0x0c, 0x09} */
void step_rst(void);                        // 声明定位函数
void delay50ms(int);                        // 声明延迟函数
```

```
main()                                          // 主程序
{   int i;                                      // 声明变量
    OUTPUT=0;                                   // 关闭输出
    step_rst();                                 // 定位
    while(1)                                    // while循环
    {  for(i=0;i<steps;i++)                     // 输出激磁信号
       {       OUTPUT=excite[i%4];              // 输出
               delay50ms(times);                // 延迟
       }                                        // 左移结束
       for(i=0;i<steps;i++)                     // 输出激磁信号
       {       OUTPUT=excite[3-i%4];            // 输出
               delay50ms(times);                // 延迟
       }                                        // 右移
    }                                           // while循环结束
}                                               // 结束主程序
//=====================================================
void step_rst(void)                             // 定位函数开始
{   char i;                                     // 声明变量
    for(i=0;i<4;i++)                            // 输出4个信号
    {  OUTPUT=excite[i];                        // 输出激磁信号
       delay50ms(20);                           // 延迟1s
    }                                           // 结束
}
//=====================================================
void delay50ms(int x)                           // 延迟函数开始
{ int i,j;                                      // 声明变量
    for(i=0;i<x;i++)                            // i循环
       for(j=0;j<600;j++); }                    // j循环
```

方向控制实验（ch10-4.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图10-23连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若有非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

● 思考一下

在本实验里，加入两个指拨开关（S1与S2），并设计一个程序，让S1接通时，则步进马达正转；让S2接通时，则步进马达反转。

10.4 实时练习



在本章里介绍步进马达的原理、步进马达的驱动电路，并探讨如何在8051里产生驱动步进马达的信号。在此请试着回答下列问题，以确认对于此部分的掌握程度。

选择题

（　）1. 下列哪种步进马达的线圈是采用中间抽头的方式？
（A）1相步进马达 （B）2相步进马达 （C）4相步进马达 （D）5相步进马达

（　）2. 某2相步进马达转子上有100齿，则其步进角度为多少？
（A）0.9° （B）1.8° （C）2° （D）4°

（　）3. 某200步的步进马达，采用1相激磁方式，需要多少个驱动信号才能旋转一周？
（A）50 （B）100 （C）200 （D）400

（　）4. 同上题，若改为采用1-2相驱动信号，需要多少个驱动信号才能旋转一周？
（A）50 （B）100 （C）200 （D）400

（　）5. 若采用ULN2003来驱动步进马达，则其最大驱动电流为多少？
（A）0.5A （B）1A （C）2A （D）3A

（　）6. 若驱动步进马达时，若需要较大电流，可使用下列哪个器件？
（A）2N3569 （B）FT5754 （C）ULN2003 （D）ULN2803

（　）7. 若步进马达的驱动信号的频率过高，会有什么现象？
（A）马达将飞脱 （B）马达将反转
（C）马达将抖动不前 （D）以上皆可能发生

（　）8. 若要使用达林顿功率晶体来驱动步进马达，可选用以下哪个？
（A）2SC1384 （B）2N2222A （C）2N3569 （D）TIP122

（　）9. 1-2相的激磁里，总共有多少个信号？
（A）4组 （B）6组 （C）8组 （D）12组

（　）10. 2相的激磁里，总共有多少个信号？
（A）4组 （B）6组 （C）8组 （D）12组

问答题

1. 试说出2相5线式步进马达与2相6线式步进马达的异同?

2. 某步进马达的转子齿间距为14.4°，则其步进角度为多少?

3. 某200步的步进马达，若以1相驱动，则每个驱动信号，将产生多少角度的位移?若改以1-2相驱动，则每个驱动信号，将产生多少角度的位移?

4. 同一个步进马达，使用1相驱动与2相驱动，有何差别?

5. 若要进行精确的位置或角度控制，在使用步进马达之前，必须进行定位或归零，这个动作是如何进行的?

6. 试问ULN2003系列驱动IC，每片IC提供多少个反相驱动器?而每个反相驱动器最大能吸取多大电流?

7. 试简述FT5754步进马达驱动IC的内部结构?

8. 请画出以FT5754驱动步进马达的电路?

9. 若要使用达林顿晶体管来驱动步进马达，它的每一极电路什么样?

10. 若程序所产生的驱动信号太快，步进马达来不及反应，结果将会怎样?

1. 编译器2的版本比编译器1的版本_____。
2. 若指定了_____，则编译器1的版本_____。
3. 若不指定_____，则_____。
4. 同一个源程序_____。
5. 若要将自己的_____。
6. 既可以_____，也可以_____。
7. 既能实现_____，又能_____。
8. 编辑器_____。
9. 若要使用_____，则_____。
10. 若指定了_____，则_____。

第11章 ADC与DAC的应用

本章内容丰富，主要包括以下两部分。

硬件部分：

模拟-数字转换原理，数字-模拟转换原理。

模拟-数字转换IC及其应用原理。

数字-模拟转换IC及其应用原理。

温度感测IC-AD590及其应用原理。

程序与实践部分：

模拟-数字转换应用程序，数字-模拟转换应用程序，配合AD-590设计一个数字温度计及简易温控装置等。

模拟（analog）信号是一种连续性的信号，大自然的很多现象（如温度、湿度、光线等）都属于这类信号；而数字（digital）信号则是一种非0即1的非连续性的信号，通常有**TTL**与**CMOS**两种电平。

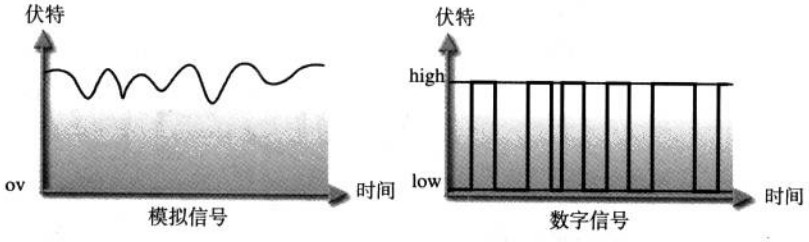


图11-1　模拟信号与数字信号

人类直接感受的就是模拟信号，不过，模拟信号比较不容易进行储存、处理与传输，而且容易失真！相反地，数字信号就比较容易储存与处理，而且比较有效率，在传输时也不易失真，是目前信号处理的主流。因此，我们就用传感器测得所要控制的模拟信号，经模拟-数字转换器（analog-digital converter，简称**ADC**）将它转换成数字信号。如此一来，就可进行较高效率的处理、储存或传输。当处理完成后，再经数字-模拟转换器（digital-analog converter，简称**DAC**）将它转换成模拟信号，来驱动控制装置（如电热器、电磁阀、马达等），这样就形成一个闭环（closed loop）的控制系统，如图11-2所示。

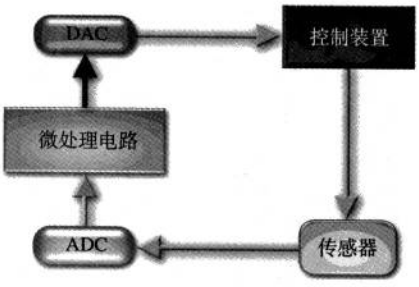


图11-2　闭环控制系统

在本单元里将介绍模拟-数字转换和数字-模拟转换及其应用。

11.1 模拟-数字转换原理



模拟-数字转换是将模拟信号变成数字信号，转换的方式说明如下。

● 并行式模拟-数字转换

并行式模拟-数字转换是用多个比较器（运算放大器）并行处理，又称为比较器型模拟-数字转换。此种模拟-数字转换以数个比较器同时检测输入的模拟信号，然后对其编

码，即可产生数字信号，如图11-3所示。

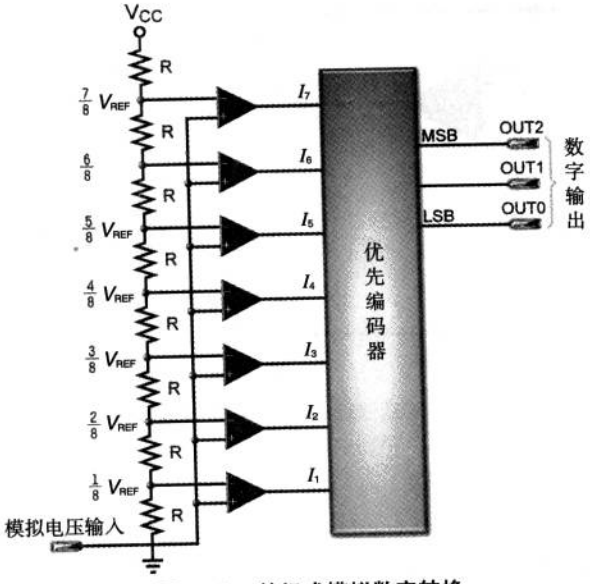


图11-3 并行式模拟数字转换

并行式模拟-数字转换的特性如下。

- 转换速度快。
- 所需要的电路较复杂，以n个位的并行式模拟-数字转换为例，则需要2^n个精密电阻、2^n-1个比较器以及一个n位的优先编码器。

连续逼近式模拟－数字转换

连续逼近式模拟-数字转换器（successive-approximation ADC）采用乘2/除2比对、快速接近的方式，将模拟信号转换成数字信号，首先将参考电压V_r与输入模拟信号比较；若输入模拟信号较高，则V_r乘以2，再与输入模拟信号比较；若输入模拟信号还是比较高，则再将V_r乘以2，与输入模拟信号比较……反之，若输入模拟信号比较低，则将V_r除以2，再与输入模拟信号比较……最后即可找到最接近的值。

对于模拟电压而言，乘以2或除以2都不容易操作。不过，对于数字信号而言，只要将数据左移一位，就是乘以2；而数据右移一位，就是除以2。移位之后的数字数据再经过数字-模拟转换，即可产生相对应的模拟信号V_r。就可与输入的模拟电压V_a相比较，以产生左移或右移的控制信号控制移位寄存器的动作，如图11-4所示。

如图11-4所示，当$V_r<V_a$时，移位寄存器将左移，而$V_r>V_a$时，移位寄存器将右移。$V_r=V_a$时，即可输出数字信号。连续逼近式模拟-数字转换的特性如下。

- n位的连续逼近式模拟-数字转换，其转换时间为n个时钟脉冲，其转换速度仅次于并行式模拟-数字转换。

• 电路较并行式模拟-数字转换的电路简单。

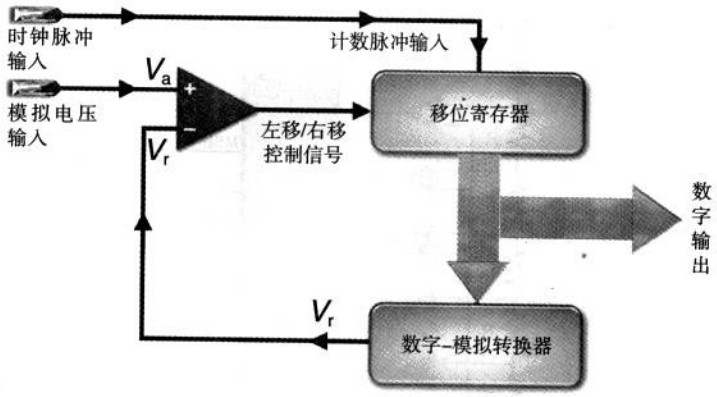


图11-4 连续逼近式模拟-数字转换概念图

连续计数式模拟－数字转换

连续计数式模拟-数字转换器（continuons counting ADC）是由比较器、上下计数器与数字-模拟转换器构成的一个闭环的转换电路，将输入的模拟信号与输出端经过数字-模拟转换器反馈回来的信号比较，以产生计数器的上数/下数控制信号，对外部输入的时钟脉冲计数，如图11-5所示。

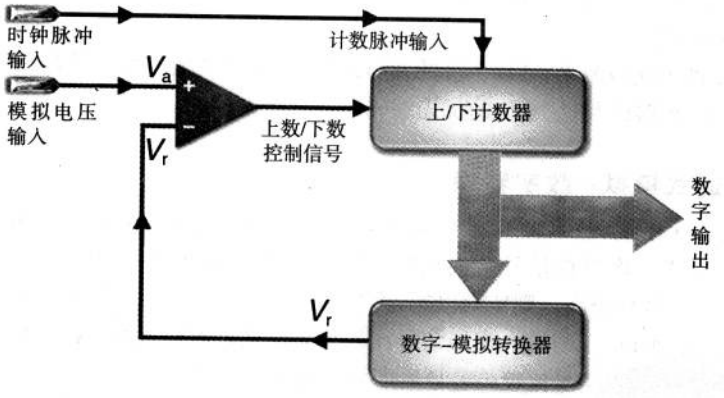


图11-5 连续计数式模拟-数字转换概念图

如图11-5所示，当$V_r<V_a$时，计数器将上数，而$V_r>V_a$时，计数器将下数。当$V_r=V_a$时，即停止计数，而输出数字信号。连续计数式模拟-数字转换的特性如下。

• 转换速度根据输入模拟电压而不同，模拟电压越高所需要的转换时间越长。

• 电路较并行式模拟-数字转换的电路简单。

双斜率式模拟－数字转换

双斜率式模拟-数字转换器（dual slope ADC）又称为积分式模拟-数字转换器，这是

定电流积分器，先用输入的模拟信号来充电，然后改用固定的参考电压进行放电，而放电期间就是计数器计数的时间。放电完毕将停止计数，而计数的结果就是所要输出的数字信号，如图11-6所示。

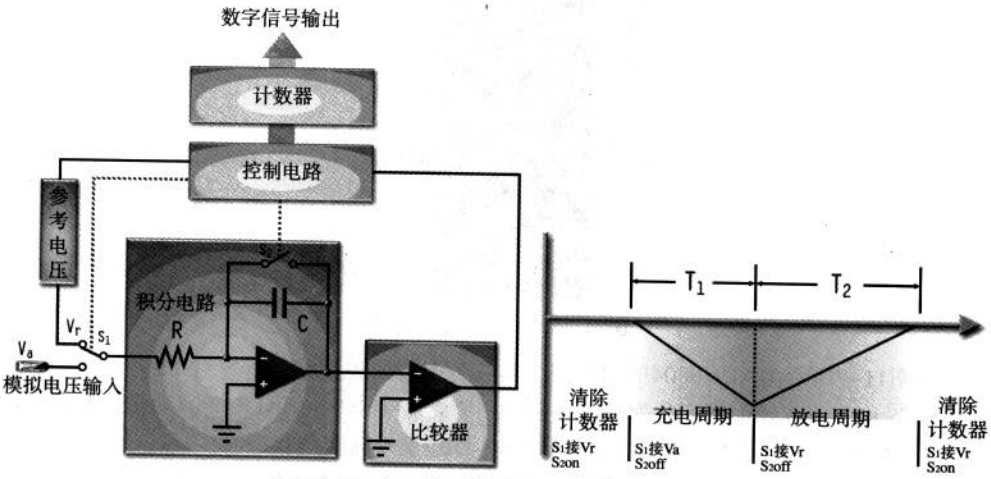


图11-6 双斜率式模拟-数字转换概念图

如图11-6所示，T_1为输入模拟电压充电所产生的斜率，T_2则为连接参考电压放电的所产生的斜率。双斜率式模拟-数字转换的特性如下。

- 转换速度最慢。
- 精密度高，稳定性佳。
- 噪声免役力良好。

11.2 认识AD转换IC



市面上的模拟-数字IC很多，而在学校里以ADC0804系列模拟-数字转换器为主，在此将以ADC0804为例，说明如下。

特性

ADC0804的特性如下。

- CMOS的连续逼近式AD转换器。
- 具有8位解析能力，转换时间为100μs，而最大误差为1个LSB值（最小电压刻度）。
- 采用差动式模拟电压输入，三态式数字输出。

引脚

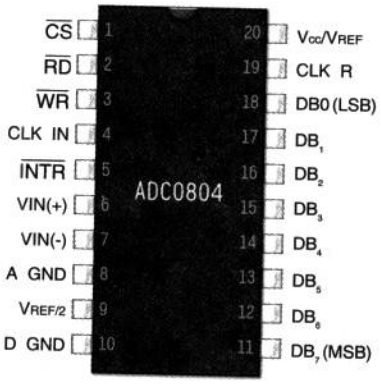


图11-7 ADC0804引脚图

如图11-7所示为ADC0804的引脚图，对其各引脚的说明如下。

- $\overline{CS}$：芯片选择引脚，此为低电平驱动引脚，若 $\overline{CS}$=0，则ADC0804动作；若 $\overline{CS}$=1，则ADC0804不动作，输出数据引脚DB0~DB7呈现高阻抗状态。
- $\overline{RD}$：数据读取引脚，此为低电平驱动引脚，若 $\overline{CS}$=0、且 $\overline{RD}$=0时，则可由DB0~DB7读取ADC0804的输出数字数据。
- $\overline{WR}$：开始转换引脚，此为低电平驱动引脚，若 $\overline{WR}$=0，即可使ADC0804开始进行模拟-数字转换操作。
- $\overline{INTR}$：完成转换引脚，此为低电平驱动引脚，若 $\overline{INTR}$=0，表示ADC0804已完成模拟-数字转换动作，而此信号常被用来通知微控制器，请它中断提取数字数据。
- **CLK IN**：时钟脉冲输入引脚，ADC0804接受100~1460kHz的时钟脉冲。而我们可配合CLK R引脚，以外加的电阻、电容，由内部电路自行产生时钟脉冲，如图11-8所示，其频率为：

$$f_{CLK} = \frac{1}{1.1RC}$$

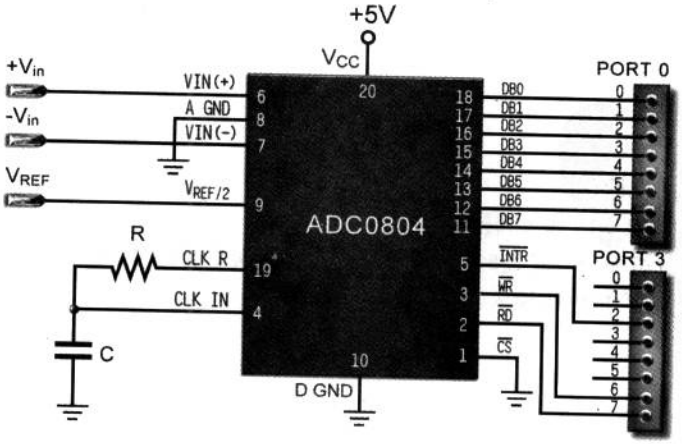


图11-8 ADC0804时钟脉冲电路

- **CLK R**：时钟脉冲输出引脚，如图11-8所示，可连接电阻以产生时钟脉冲。
- $V_{REF/2}$：参考电压输入引脚。通常本引脚所连接的电压，是输入模拟电压最大值的一半。
- V_{IN+}：模拟电压输入的正极引脚，所输入的模拟电压不得超过$V_{REF/2}$引脚的电压。
- V_{IN-}：模拟电压输入的负极引脚。
- V_{CC}：电源引脚或参考电压引脚，通常是连接+5V，以作为电源之用。若$V_{REF/2}$引脚没有连接参考电压时，则ADC0804以本引脚上的电压为参考电压。
- **D GND**：数字信号接地引脚。
- **A GND**：模拟信号接地引脚，通常本引脚都与D GND引脚连接后接地，若处理高干扰性的模拟信号，本引脚自可单独接地。
- **DB0~DB7**：数字输出数据引脚，这8只引脚为三态式输出，可直接连接微控制器的数据总线，若此IC不输出时，则这8只引脚呈现高阻抗状态。

操作方式

ADC0804的操作方式说明如下：

- 连续转换：ADC0804最简单的操作方式，就是让它不停地进行转换。如图11-9所示，$\overline{CS}$与$\overline{RD}$引脚连接到接地端，再将$\overline{INTR}$引脚连接到$\overline{WR}$引脚，如此就可令$\overline{INTR}$引脚输出的完成转换信号，成为$\overline{WR}$引脚的开始转换信号。而微控制器随时可读取这个数据总线上的数据。

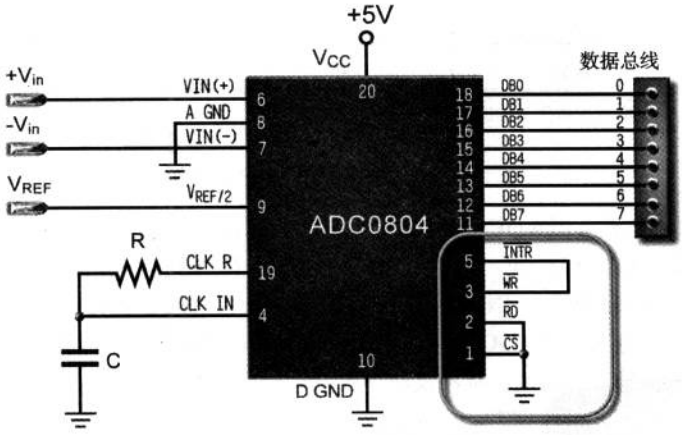


图11-9 ADC0804的连续转换电路

- 交互式控制：如图11-10所示，将$\overline{CS}$引脚接地、将$\overline{WR}$与$\overline{RD}$引脚连接到微控制器的输出端口，此信号称为**START**或**SOC**（start of convert），若微控制器通过这个输出口输出一个负脉冲，则ADC0804即可进行模拟-数字转换。当ADC0804完成转换后，则由$\overline{INTR}$引脚输出一个低电平的脉冲，此信号称为IRQ，若将这个信号连接到微控制器的输入口，则该微控制器将可以查询方式或中断方式检测到这个信号，于是进行ADC0804数字数据的读取；若将这个信号连接到微控制器的外部中断引脚，则该微控制器将中断，以进行ADC0804数字数据的读取。

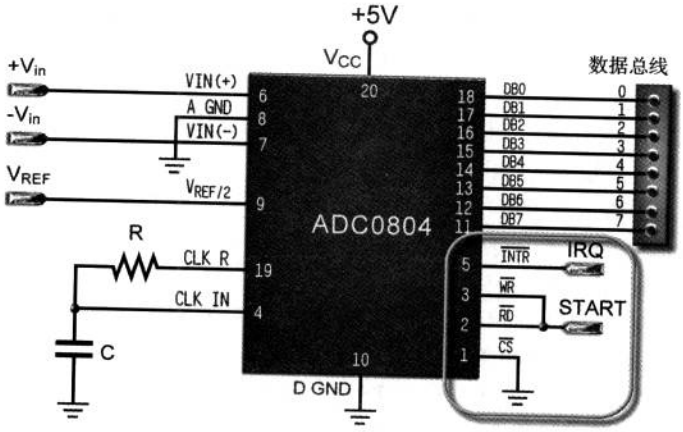


图11-10　ADC0804的交互式控制电路

ADC0804的操作时序如图11-11所示。

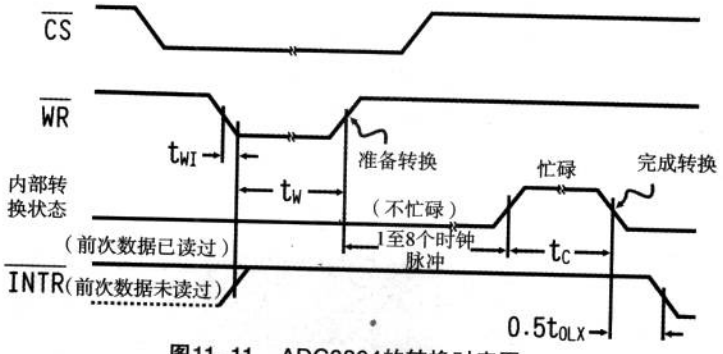


图11-11　ADC0804的转换时序图

如图11-11所示，当ADC0804的 $\overline{CS}$ 引脚为低电平，且 $\overline{WR}$ 引脚也为低电平时，ADC0804内部开始进行转换，转换期间，$\overline{INTR}$ 引脚为高电平。如图11-12所示，当ADC0804内部转换完成后，$\overline{INTR}$ 引脚转为低电平。这时候，若 $\overline{CS}$ 引脚又变成低电平，且 $\overline{RD}$ 引脚也为低电平，则ADC0804转换的结果，将随 $\overline{INTR}$ 引脚转为高电平时，放入数据总线（DB0-DB7），以供微控制器读取。

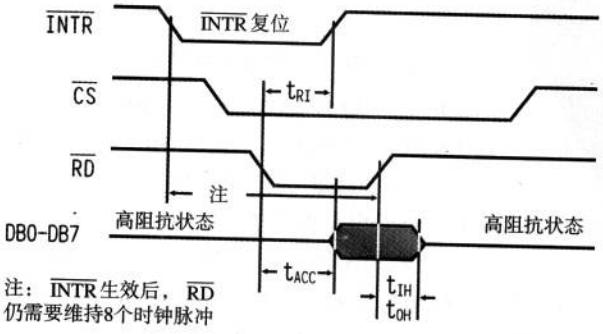


图11-12　ADC0804的时序图

整个交互的控制如图11-13所示，第一步由微控制器送一个低电平 $\overline{WR}$ 的信号到ADC0804，以启动ADC0804；当ADC0804转换完成后，即进入第二步，ADC0804送出一个低电平的 $\overline{INTR}$ 信号，请微控制器来提取；第三步，当微控制器要来提取之前，送一个低电平的 $\overline{RD}$ 信号通知ADC0804；第四步，微控制器即可读取总线的数据。

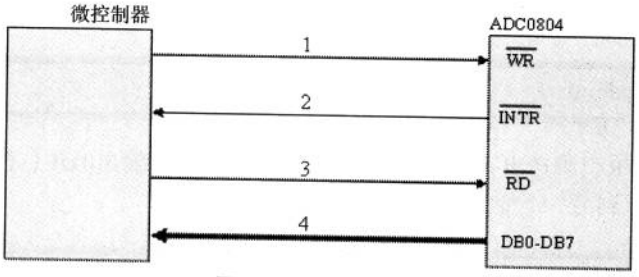


图11-13 交互信号

8051与ADC0804的连接

一般8051与ADC0804的连接可分为控制线与数据线，若ADC0804采用连续转换的方式（如图11-9所示），则不需要连接控制线，直接将DB0~DB7连接到8051的任一个PORT即可。在程序方面，也把连接的输入/输出口当成一般的输入口，随时读取其中的数据。若ADC0804采用交互式控制，则除了数据线外，还需要将其START及IRQ控制线（如图11-10所示）各连接到8051输入/输出端口的任意一位，例如P2.0、P2.1等。我们可将ADC0804的 $\overline{WR}$ 引脚连接到8051的 $\overline{WR}$ 引脚（P3.6），将ADC0804的 $\overline{RD}$ 引脚连接到8051的 $\overline{RD}$ 引脚（P3.7），将ADC0804的 $\overline{INTR}$ 引脚连接到8051的INT0引脚（P3.2），而DB0~DB7连接到8051的PORT 0，如图11-14所示。

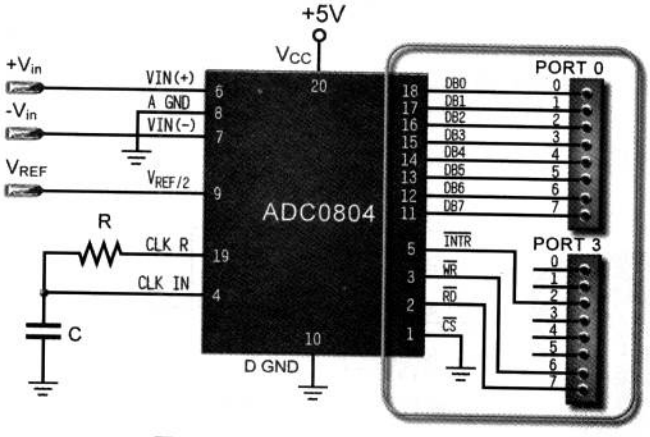


图11-14 ADC0804与8051连接

当我们要进行8051的外部存储器操作时，若使用汇编语言，则可以使用外部存储器存取指令——MOVX；而在C语言里并没有独立的存取外部存储器的指令，而是以“xdata”存储器形式作为操作外部存储器的依据。只要将某个数据变量声明为“xdata”存储器形式，则将该变量看作一个外部存储器，只要动到该数据，就会触动外部存储器

的操作。例如要声明一个8位的“xdata”存储器形式变量adc，使用以下语句：

```
unsigned char xdata  adc;
```

此后，若要将某一数据放入adc变量，使用以下语句：

```
adc=0xff;
```

则8051的 $\overline{WR}$ 引脚送出一个低电平信号，同时Port 0将输出0xff（不重要）。同样地，若要将adc变量存到另一个变量，使用以下语句。

```
results=adc;
```

则8051的 $\overline{RD}$ 引脚送出一个低电平信号，同时，将读取Port 0的数据。

在进行ADC0804的控制时，当然先要声明“xdata”存储器形式的变量，紧接着的第一步是由8051通过 $\overline{WR}$ 引脚，送出一个低电平的启动信号，这时候就可以利用刚才所介绍的“adc=0xff;”指令，以启动ADC0804。当ADC0804完成转换后，将通过 $\overline{INTR}$ 引脚通知8051；而这支引脚连接到8051的INT0引脚（P3.2），我们可利用INT 0中断的方式，在中断子程序里，以刚才所介绍的“results=adc;”指令，将ADC0804转换的结果，存入results变量。

11.3 数字-模拟转换原理



一般数字-模拟转换器（digital-analog converter，简称DAC）是由电阻网络所构成的，常见的数字-模拟转换电路有加权电阻网络及R-2R电阻网络两种。

加权电阻网络

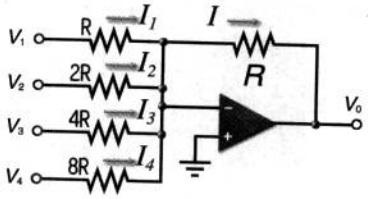


图11-15　加权电阻网络

如图11-15所示为加权电阻网络，其中各电流如下：

$$I_1=\frac{V_1}{R}、\quad I_2=\frac{V_2}{2R}、\quad I_3=\frac{V_3}{4R}、\quad I_4=\frac{V_4}{8R}$$

$I=I_1+I_2+I_3+I_4$

$$
\begin{aligned}
V_o &= -IR \\
&= -(I_1 + I_2 + I_3 + I_4)R \\
&= -(\frac{V_1}{R} + \frac{V_2}{2R} + \frac{V_3}{4R} + \frac{V_4}{8R})R \\
&= -(V_1 + \frac{1}{2}V_2 + \frac{1}{4}V_3 + \frac{1}{8}V_4) \\
&= -\frac{1}{8}(8V_1 + 4V_2 + 2V_3 + V_4) \\
&= -\frac{1}{8}(2^3V_1 + 2^2V_2 + 2^1V_3 + 2^0V_4)
\end{aligned}
$$

其中的V_1、V_2、V_3、V_4分别为数字数据的bit 3、bit 2、bit 1及bit 0，其电压值非0V就是5V，而$-\frac{1}{8}$可由运算放大器的反馈电阻来调整其大小。若此电路输入1111数字数据，则输出电压V_O为：

$$
\begin{aligned}
V_o &= -(5 + \frac{1}{2} \times 5 + \frac{1}{4} \times 5 + \frac{1}{8} \times 5) \\
&= -(5 + 2.5 + 1.25 + 0.625) \\
&= -9.375
\end{aligned}
$$

同理，若输入1110数字数据，则输出电压V_O为：

$$
\begin{aligned}
V_o &= -(5 + \frac{1}{2} \times 5 + \frac{1}{4} \times 5 + \frac{1}{8} \times 0) \\
&= -(5 + 2.5 + 1.25 + 0) \\
&= -8.75
\end{aligned}
$$

表11-1所示为这个网络的输入/输出关系。

表11-1

bit3	bit2	bit1	bit0	V_o	bit3	bit2	bit1	bit0	V_o
0	0	0	0	0	1	0	0	0	-5
0	0	0	1	-0.625	1	0	0	1	-5.625
0	0	1	0	-1.25	1	0	1	0	-6.25
0	0	1	1	-1.875	1	0	1	1	-6.875
0	1	0	0	-2.5	1	1	0	0	-7.5
0	1	0	1	-3.125	1	1	0	1	-8.125
0	1	1	0	-3.75	1	1	1	0	-8.75
0	1	1	1	-4.375	1	1	1	1	-9.375

由上述可得知加权电阻网络数字-模拟转换的原理，在此将其特性归纳如下。

- 电路结构简单，但不容易制作，因为其中所使用的电阻值，种类太多，差异过大。

在IC的内部电路里，很难作出这样的电路。

- 由于最大与最小的电阻差异太大，非常容易造成误差，以8位的转换电路为例，其中最大电阻为最小电阻的256倍，若电阻的误差为1%，则最大电阻的误差值就比最小电阻或次小电阻还大了！所以，很难达到较高的精确度。

R－2R电阻网络

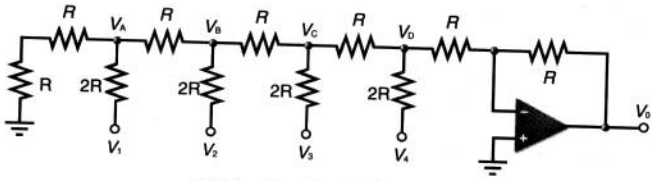


图11-16　R-2R电阻网络

如图11-16所示为R-2R电阻网络，电路结构很有规律，在此将以重迭定理与戴维宁等效电路来解析V_D的电压。

1. 只考虑V_1，则V_2、V_3、V_4视为接地，如图11-17所示。

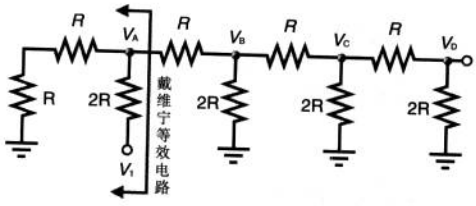


图11-17　戴维宁等效电路

由V_A向左看的戴维宁等效电路为：

$$R_{TH} = (R + R)//2R$$

$$V_{TH} = V_1 \times \frac{2R}{4R} = \frac{1}{2}V_1$$

对电路进行修改，如图11-18所示。

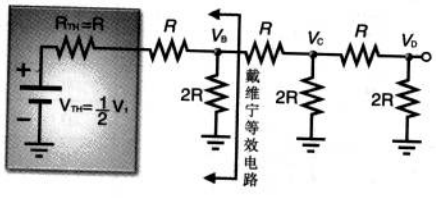


图11-18　修改后的电路1

由V_B向左看的戴维宁等效电路为：

$$R_{TH} = (R + R)//2R$$

$$V_{TH} = \frac{1}{2}V_1 \times \frac{2R}{4R} = \frac{1}{4}V_1$$

对电路进行修改，如图11-19所示。

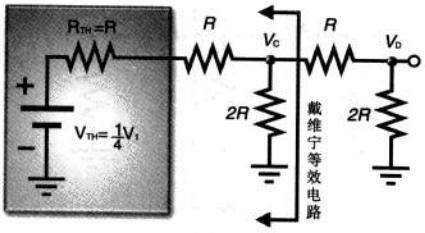


图11-19 修改后的电路2

由V_C向左看的戴维宁等效电路为：

$$R_{TH} = (R + R)//2R$$

$$V_{TH} = \frac{1}{4}V_1 \times \frac{2R}{4R} = \frac{1}{8}V_1$$

对电路进行修改，如图11-20所示。

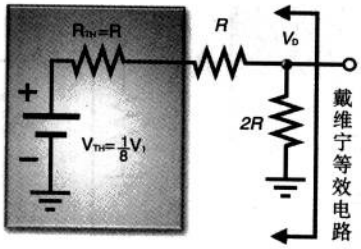


图11-20 修改后的电路3

由V_D向左看的戴维宁等效电路为：

$$R_{TH} = (R + R)//2R$$

$$V_{TH} = \frac{1}{8}V_1 \times \frac{2R}{4R} = \frac{1}{16}V_1$$

对电路进行修改，如图11-21所示。

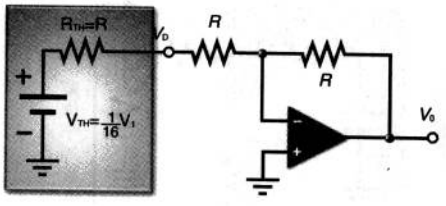


图11-21 修改后的电路4

所以，

$$V_O = -\frac{R}{2R} \times \frac{1}{16}V_1 = -\frac{1}{32}V_1$$

2. 只考虑V_2，则V_1、V_3、V_4视为接地，如图11-22所示。

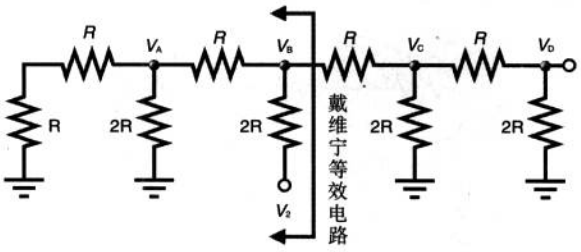


图11-22　戴维宁等效电路

由V_B向左看的戴维宁等效电路为：

$$R_{TH} = ((R + R)//2R + R)//2R = R$$

$$V_{TH} = V_2 \times \frac{2R}{4R} = -\frac{1}{2}V_2$$

对电路进行修改，如图11-23所示。

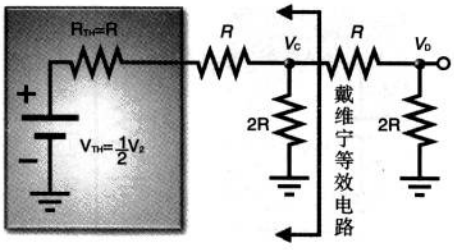


图11-23　修改后的电路1

由V_C向左看的戴维宁等效电路为：

$$R_{TH} = (R + R)//2R$$

$$V_{TH} = \frac{1}{2}V_2 \times \frac{2R}{4R} = \frac{1}{4}V_2$$

对电路进行修改，如图11-24所示。

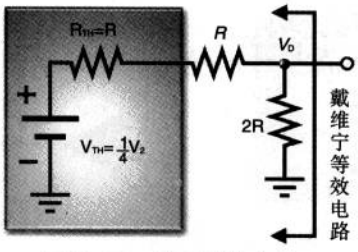


图11-24　修改后的电路2

由V_D向左看的戴维宁等效电路为：

$$R_{TH} = (R + R)//2R$$

$$V_{TH} = \frac{1}{4}V_2 \times \frac{2R}{4R} = \frac{1}{8}V_2$$

对电路进行修改，如图11-25所示。

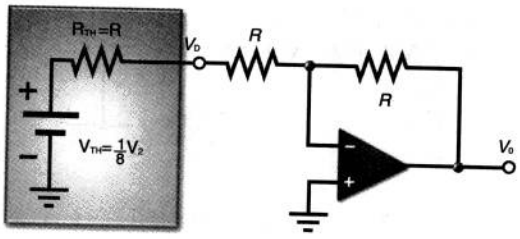


图11-25 修改后的电路3

所以，

$$V_O''=-\frac{R}{2R}\times\frac{1}{8}V_2=-\frac{1}{16}V_2$$

3. 只考虑V_3，则V_1、V_2、V_4视为接地，如图11-26所示。

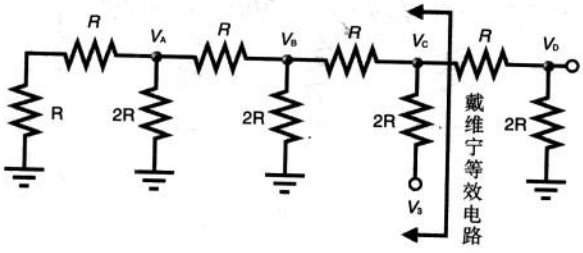


图11-26 戴维宁等效电路

由V_C向左看的戴维宁等效电路为：

R_{TH} = (((R + R)//2R + R)//2R + R)//2R = R

$$V_{TH}=V_3\times\frac{2R}{4R}=\frac{1}{2}V_3$$

对电路进行修改，如图11-27所示。

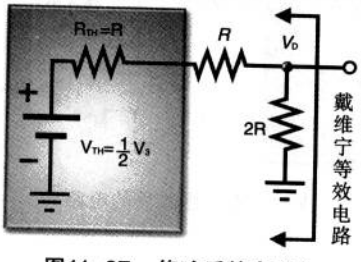


图11-27 修改后的电路1

由V_D向左看的戴维宁等效电路为：

R_{TH} = (R + R)//2R

$$V_{TH}=\frac{1}{2}V_3\times\frac{2R}{4R}=\frac{1}{4}V_3$$

对电路进行修改，如图11-28所示。

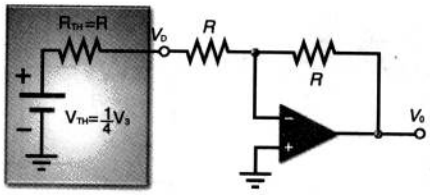


图11-28 修改后的电路2

所以，

$$V_O''' = -\frac{R}{2R} \times \frac{1}{4} V_3 = -\frac{1}{8} V_3$$

4. 只考虑V_4，则V_1、V_2、V_3视为接地，如图11-29所示。

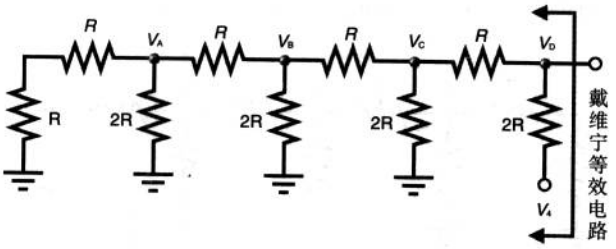


图11-29 戴维宁等效电路

由V_D向左看的戴维宁等效电路为：

$$R_{TH} = ((((R + R)//2R + R)//2R + R)//2R + R)2R = R$$

$$V_{TH} = V_4 \times \frac{2R}{4R} = \frac{1}{2} V_4$$

对电路进行修改，如图11-30所示。

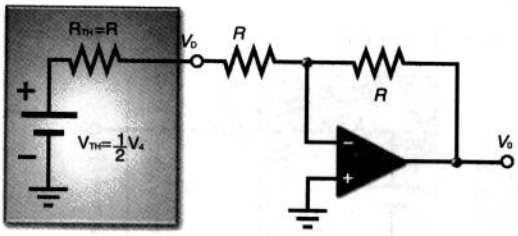


图11-30 修改后的电路1

所以，$V_O'''' = -\frac{R}{2R} \times \frac{1}{2} V_4 = -\frac{1}{4} V_4$

综合前4项：

$$V_O' = -\frac{R}{2R} \times \frac{1}{16} V_1 = -\frac{1}{32} V_1$$

$$V_O^{'}=-\frac{R}{2R}\times\frac{1}{8}V_2=-\frac{1}{16}V_2$$

$$V_O^{''}=-\frac{R}{2R}\times\frac{1}{4}V_3=-\frac{1}{8}V_3$$

$$V_O^{'''}=-\frac{R}{2R}\times\frac{1}{2}V_4=-\frac{1}{4}V_4$$

，根据重迭定理可得：

$$\begin{aligned}V_O&=V_O^{'}+V_O^{''}+V_O^{'''}+V_O^{''''}\\&=-(\frac{1}{32}V_1+\frac{1}{16}V_2+\frac{1}{8}V_3+\frac{1}{4}V_4)\\&=-\frac{1}{32}(V_1+2V_2+4V_3+8V_4)\\&=-\frac{1}{32}(2^0V_1+2^1V_2+2^2V_3+2^3V_4)\end{aligned}$$

其中的V_1、V_2、V_3、V_4分别为数字数据的bit 0、bit 1、bit 2及bit 3，其电压值非0V就是5V，而$-\frac{1}{32}$可由运算放大器的反馈电阻来调整其大小。若此电路输入1111数字数据，则输出电压V_O为：

$$\begin{aligned}V_O&=-\frac{1}{32}(5+10+20+40)\\&=-\frac{75}{32}\\&=-2.345375\end{aligned}$$

由上述可得知R-2R电阻网络数字-模拟转换的原理，在此对其特性归纳如下。

- 电路结构简单，其中的电阻值只有两种，不管是自制电路还是IC的内部电路里，都很容易实现这样的电路。
- 不管是ADC还是DAC，其电压分辨率V_{RES}（或V_{LSB}）与其数字的位数及电压范围（参考电压）有关，如下：

$$V_{RES}=\frac{V_{REF}}{2^n-1}=\frac{V_{max}-V_{min}}{2^n-1}$$

其中的V_{REF}就是参考电压，V_{max}为最大电压，V_{min}为最小电压，n为位数。

11.4 认识DA转换IC



市面上的数字-模拟转换IC不少，而在学校里以DAC-08系列（1408）为主，下面以DAC-08为例进行说明。

特性

DAC08的特性如下。

- 电流型R-2R电阻网络的DA转换器。
- 具有8位解析能力，转换时间为300ns。
- 电源可采用 ± 15V双电源，或+5~+15单电源。

引脚

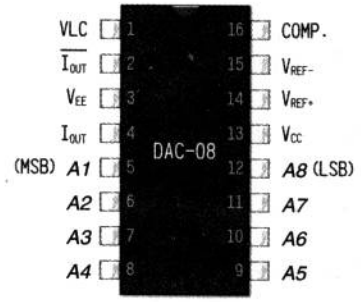


图11-31　DAC-08引脚图

如图11-31所示为DAC-08的引脚图，对其中各引脚说明如下。

- VLC：临界电压控制输入引脚，其功能是设定数字信号电平，接地即可。
- $\overline{I_{OUT}}$ ：互补模拟电流输出引脚，$\overline{I_{OUT}} = I_{FS} - I_{OUT}$ ，其中的I_{OUT}为模拟输出电流，I_{FS}为满刻度电流（在0.2~4mA之间）

$$I_{FS} = \frac{V_{REF}}{R_{REF}} \times \frac{255}{256}$$

- V_{EE}：负电源引脚，其电压范围为-4.5~-18V。
- I_{OUT}：模拟电流输出引脚，而

$$I_{OUT} = \frac{2^{n-1} \cdot D_{n-1} + 2^{n-2} \cdot D_{n-2} + ... + 2^0 \cdot D_0}{2^n} \times I_{REF}$$

其中 $I_{REF} = \frac{V_{REF}}{R_{REF}}$

$$V_O = -I_{OUT} \times R_O = -\frac{2^{n-1} \cdot D_{n-1} + 2^{n-2} \cdot D_{n-2} + ... + 2^0 \cdot D_0}{2^n} \times I_{REF} \times R_O$$

- A1~A8：这8只引脚为数字输出引脚，其中A1为最高位（MSB）、A8为最低位（LSB）。
- V_{CC}：正电源引脚，其电压范围为+4.5~+18V。
- V_{REF+}：正参考电压输入引脚。
- V_{REF-}：负参考电压输入引脚。
- COMP.：补偿引脚，外接补偿电容，以避免高频振荡。

操作方式

如图11-32所示为DAC-08的基本电路，其中将正电源连接+5V，负电源接地，$I_{REF}=V_{REF}/R_{REF}$，若希望I_{REF}为1mA，而V_{REF}连接+5V的话，则R_{REF}可采用5kΩ即可。

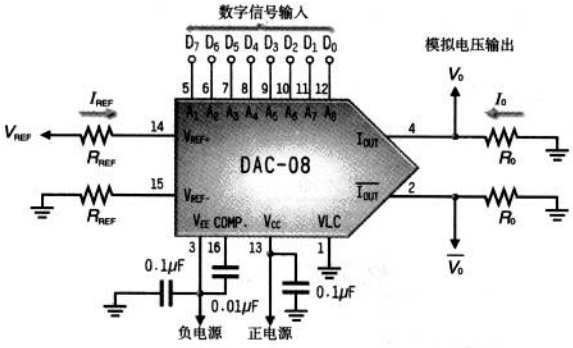


图11-32 基本电路

另外，由于

$$I_{OUT}=-\frac{2^{n-1}\cdot D_{n-1}+2^{n-2}\cdot D_{n-2}+\ldots+2^{0}\cdot D_{0}}{2^{n}}\times I_{REF}$$，所以

$$V_O=-I_{OUT}\times R_O=-\frac{2^{n-1}\cdot D_{n-1}+2^{n-2}\cdot D_{n-2}+\ldots+2^{0}\cdot D_{0}}{2^{n}}\times I_{REF}\times R_O$$，

若使RO也为5kΩ，则

$$I_{OUT}=-\frac{2^{7}\cdot D_{7}+2^{6}\cdot D_{6}+\ldots+2^{0}\cdot D_{0}}{256}\times 1\text{m}$$

$$V_O=-\frac{2^{7}\cdot D_{7}+2^{6}\cdot D_{6}+\ldots+2^{0}\cdot D_{0}}{256}\times 1\text{m}\times 5\text{k}$$

$$=-\frac{2^{7}\cdot D_{7}+2^{6}\cdot D_{6}+\ldots+2^{0}\cdot D_{0}}{256}\times 5$$

若数字信号输入为11111111，则模拟电压输出为：

$$I_{OUT}=\frac{2^{7}\cdot 1+2^{6}\cdot 1+\ldots+2^{0}\cdot 0}{256}\times 1\text{m}=\frac{255}{256}\text{m}\cong 0.996\text{mA}$$

$$V_O=-0.996\text{m}\times 5\cong -4.98\text{V}$$

若数字信号输入为11111110，则模拟电压输出为：

$$I_{OUT}=\frac{2^{7}\cdot 1+2^{6}\cdot 1+\ldots+2^{0}\cdot 0}{256}\times 1\text{m}=\frac{254}{256}\text{m}\cong 0.992\text{mA}$$

$$V_O=-0.992\text{m}\times 5\cong -4.96\text{V}$$

同理，若数字信号输入为00000001，则模拟电压输出为：

$$I_{OUT} = \frac{1}{256} m \cong 0.0039mA$$

$$V_o = -0.0039m \times 5 \cong -0.02V$$

由DAC-08直接输出的话，由于输出阻抗较高，容易造成负载效应，所以在其输出端加一个运算放大器，如图11-33所示（其中运算放大器的电源可采用 ± 12V），即可得到较佳的输出结果，而其电压转换的结果，除没有的 $\overline{V_o}$ 输出外，与原电路相同。若要有 $\overline{V_o}$ 与 $\overline{V_o}$ 对称性输出，则可对电路进行修改，如图11-34所示。

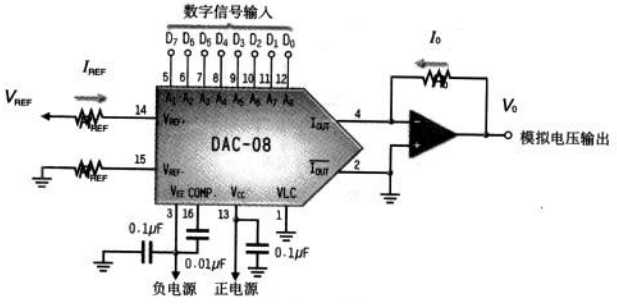


图11-33 低阻抗输出电路

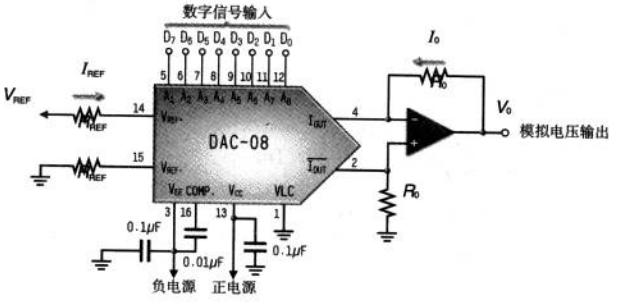


图11-34 对称性输出电路

8051与DAC-08的连接

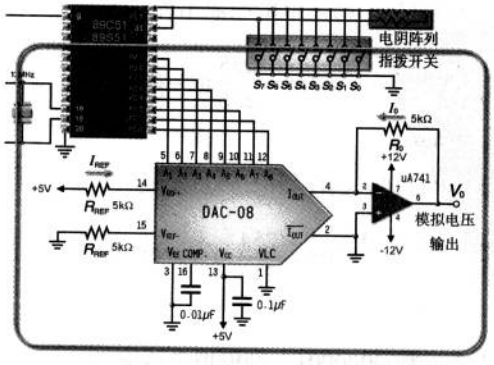


图11-35 DAC-08与8051连接

如图11-35所示，8051与DAC-08的连接只是简单的总线连接而已，例如把DAC-08的A1~A8连接到8051的Port 2（或其他输入/输出口），就可把8051由Port 2输出的数字数据转换成模拟电压。

11.5 内含ADC的51系列



大部分提供51系列的半导体厂商都有提供内含ADC的51单芯片，以Atmel公司为例，其内含ADC的51单芯片有AT89C5115、AT89C51AC2及AT89C51AC3等，而这三个单芯片除内含10位ADC外，其内部结构也比标准的89C51强很多！下面简要介绍这几种单芯片。

AT89C5115

- RAM：256 bytes RAM、256 bytes XRAM。
- ROM：16k bytes Flash ROM。
- 14个中断源。
- 3个16位计时计数器。
- 1个全双工UART。
- 最高工作频率40MHz。
- 输入/输出口：16或20条数位I/O，视包装而定。
- 双信道16位的PCA，可作为8位脉冲宽度调变（Pulse Width Modulation，简称**PWM**）。
- 两组数据指针寄存器。
- 21位看门狗定时器。
- 10位ADC。
- 提供Power-Down及Idle等两种省电模式。
- 电源范围：3~5.5V。
- 器件包装：SOIC28、SOIC24、PLCC28、VQFP32，在学校或训练单位可采用PLCC28，配合PLCC脚座，以方便练习。

AT89C51AC2

- RAM：256 bytes RAM、1k bytes XRAM。
- ROM：32k bytes Flash ROM。
- 14个中断源。
- 3个16位计时计数器。
- 1个全双工UART。
- 最高工作频率：40M Hz。
- 输入/输出口：34条数位I/O，视包装而定。

- 双信道16位的PCA，可作为8位脉冲宽度调变（Pulse Width Modulation，简称**PWM**）。
- 两组数据指针寄存器。
- 21位看门狗定时器。
- 10位ADC。
- 芯片内置仿真器逻辑，即On-chip Emulator Logic。
- 提供Power-Down及Idle等两种省电模式。
- 电源范围：3~5.5V。
- 器件包装：PLCC44、VQFP44，在学校或训练单位可采用PLCC44，配合PLCC脚座，以方便练习。

AT89C51AC3

- RAM：256 bytes RAM、2k bytes ERAM。
- ROM：32k bytes Flash ROM。
- 14个中断源。
- 3个16位计时计数器。
- 1个全双工UART。
- 最高工作频率：60MHz。
- 输入/输出口：36条数位I/O，视包装而定。
- 双信道16位的PCA，可作为8位脉冲宽度调变（Pulse Width Modulation，简称**PWM**）。
- 两组数据指针寄存器。
- 21位看门狗定时器。
- 10位ADC。
- SPI接口（SPI为Serial Peripheral Interface的简称）是连接Motorola公司所发展的同步串行数据连接标准（synchronous serial data link standard），用以控制串行外围装置接口总线（即Serial Peripheral Interface Bus）的控制器，如图11-36所示为SPI总线系统：

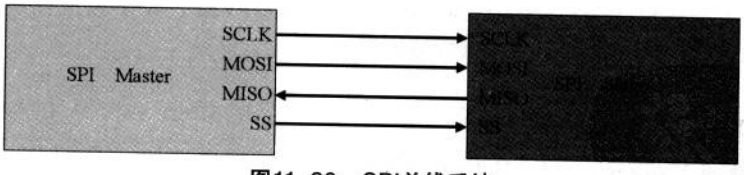
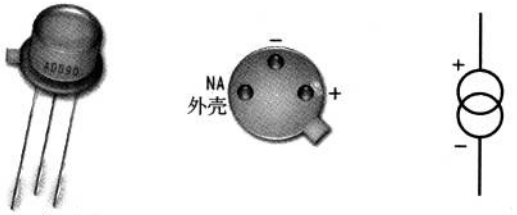


图11-36 SPI总线系统

- 芯片内置仿真器逻辑，即On-chip Emulator Logic。
- 提供Power-Down及Idle等两种省电模式。
- 电源范围：3~5.5V。
- 器件包装：PLCC44、VQFP44、VQFP64、PLCC52，在学校或训练单位可采用PLCC44或PLCC52，配合PLCC脚座，以方便练习。

11.6 认识温度传感器



AD590外观图　　AD590底视图　　AD590符号

图11-37　AD590的外观、底部引脚图与符号

美国Analog Device公司所开发的AD590是体积小、使用方便的温度传感器。如图11-37所示，AD590就像一般小型金属壳包装的晶体管，同样是三只引脚，很容易被误以为是晶体管。实际上，它是一个温度感测IC，而且是不便宜的小器件。虽然AD590有三只引脚，通常只使用其中两只引脚，其特性如下。

- 其输出电流与开氏（开尔文）温度成正比，开氏温度0°时输出0A，开氏温度每上升1°电流增加1μA（即1μA/K）。其中的开氏温度（Kelvin temperature scale）又称为绝对温度（absolute temperature scale），而开氏温度与摄氏温度（Celsius temperature scale）的关系为开氏温度等于摄氏温度加上273。换言之，摄氏温度每上升1°AD590电流增加1μA。
- 有效温度感测范围为-55~150℃。
- 可采用的电源范围为4~30V。

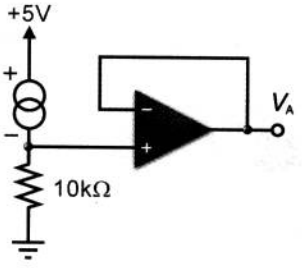


图11-38　AD590连线

如图11-38所示，最简单的AD590接口是串接一个10kΩ电阻再接地，即可产生10×（273.2+T℃）mV，这个电压先经一个运算放大器所组成的缓冲器，以避免负载效应。当0℃时，V_A=10×273.2mV=2.732V、100℃时，V_A=10×373.2mV=3.732V，不是很人性化，如果将V_A减去2.732，则0℃时V_A=0V、100℃时V_A=1V，每增加1℃，V_A增加0.01V（即10mV），这样比较容易被接受！在此利用一个运算放大器，以进行减法功能，如图11-39所示。

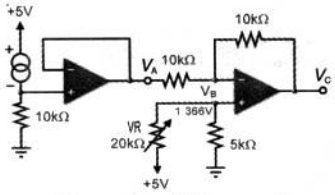


图11-39 减去2.732V

如图11-39所示，请用数字电表测量V_B，调整VR半固定电阻，让V_B为1.366V，则$V_C=-V_A+2V_B=-(V_A-2.732)$。

若要使用前述的ADC0804将此电压转换成数字信号，而ADC0804所采用的参考电压V_{REF}为2.5V的话，其V_{LSB}约为0.0196V（接近0.02V）。则还需将图11-39中的V_C再放大-2倍，使温度增加1℃时V_C增加0.02V，如图11-40所示。

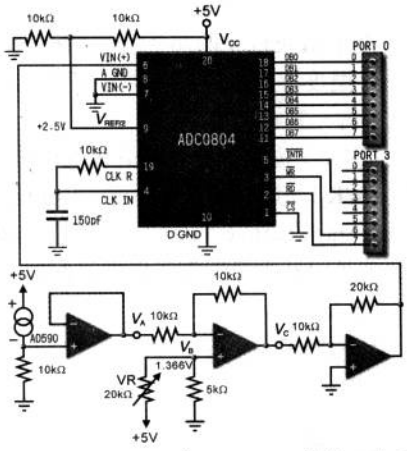


图11-40 AD590与ADC0804的接口电路

11.7 实例演练



在本单元里提供了如下4个实例。

11.7.1 ADC连续转换实例演练

● 实验要点

如图11-41所示，ADC0804连接到8051的Port 2，而ADC0804的$\overline{WR}$引脚与$\overline{INTR}$引脚相连接，而$\overline{RD}$与$\overline{CS}$引脚都接地，如此就能使ADC0804不断地进行转换。转换的结果也随时放在总线上，8051可从Port 2读取它。ADC0804的$+V_{in}$引脚连接到10kΩ可变电阻，调整该可变电阻，即可改变其电压值，可从0V调整到5V。$V_{REF/2}$引脚连接到一个分压电路，以取得2.5V的参考电压，如此将决定所要测量的电压最大值不得超过5V，即$V_{REF/2}\times 2$。

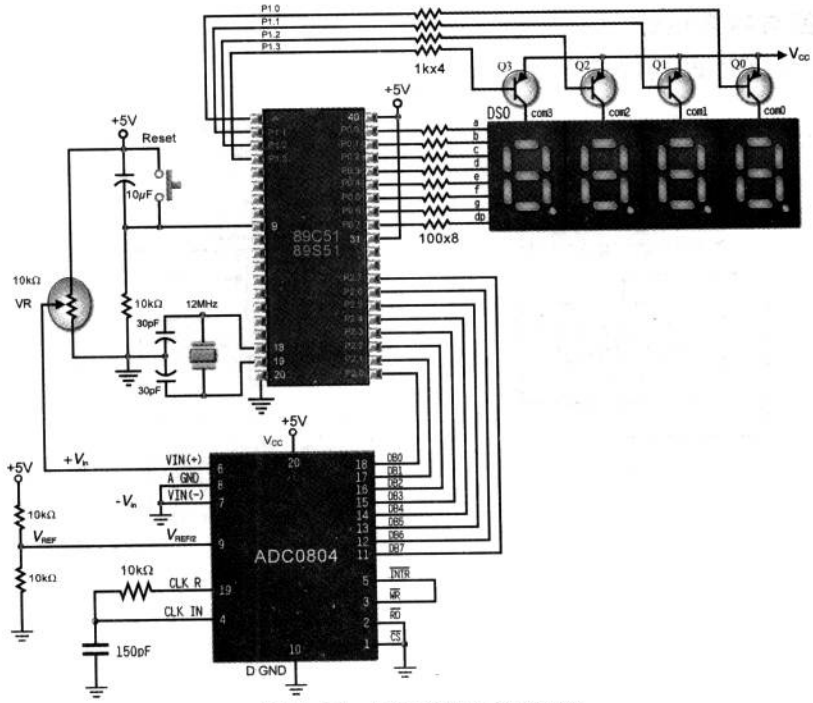


图11-41 ADC连续转换电路图

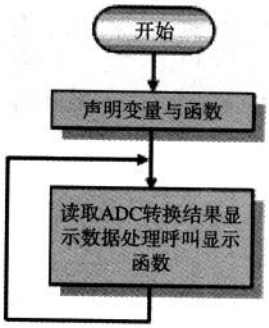
8051除了读取ADC0804的转换结果外，也将其值做适当处理后显示在4数字的7段LED数码管模块上，如此即为一个简易的数字电压表。当ADC0804的$+V_{in}$引脚连接到5V时，DB7 – DB0将输出11111111，即0xff；若$+V_{in}$引脚连接到0V时，DB7 – DB0将输出00000000，即0x00，其分辨率为 $\frac{5-0}{255}=\frac{1}{51}\cong 0.0196\text{V}$。

当测量到5V时，ADC0804转换的结果为11111111（即255），我们希望7段LED数码管上显示5000，所以要把255 × 19.6，才会得到5000；在此将乘以196，而非19.6，如此将可得到10倍大的值，即50000。将此值（results）除以10000，所得的商数输出到7段LED数码管的千位数；将results除以1000，其商数再除以10，所得的余数输出到7段LED数码管的百位数；将results除以100，其商数再除以10，所得的余数输出到7段LED数码管的十位数；将results除以10，所得的余数除以10，其商数输出到7段LED数码管的个位数，如下所示：

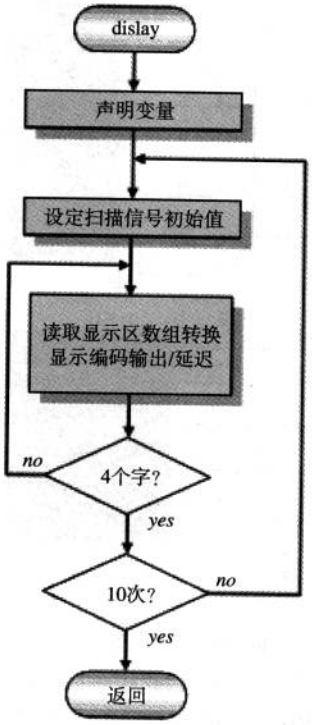
```
results= adc*196;                    // 乘以196倍
disp[3]=results/10000;               // 取得千位数
disp[2]=(results/1000)%10;           // 取得百位数
disp[1]=(results/100)%10;            // 取得十位数
disp[0]=(results/10)%10;             // 取得个位数
```

流程图与程序设计

流程图与整个程序设计如下所示：



主程序




```
/* ADC连续转换实例演练(ch11-1.c) */
#include <reg51.h>
/*声明驱动信号数组*/
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,
               0x92, 0x83, 0xf8, 0x80, 0x98 };
//=========================================
unsigned char disp[4]={0, 0, 0, 0};      // 声明显示区数组
void display(void);                      // 声明显示函数
void delay1ms(char);                     // 声明延迟函数
main()                                   // 主程序
{   int results;                         // 声明变量
    while(1)                             // while开始
    {   P2=0xff;
        results= P2*196;                 // 读取ADC0804转换结果乘以196倍
        disp[3]=results/10000;           // 取得千位数
```

```
        disp[2]=(results/1000)%10;      // 取得百位数
        disp[1]=(results/100)%10;       // 取得十位数
        disp[0]=(results/10)%10;        // 取得个位数
        display();                      // 调用显示函数
    }                                   // while结束
}                                       // 主程序结束
//====显示函数====
void display(void)
{   char j,scan;                        // 声明变量
    char i=10;                          // 扫描10次
    while(--i>=0)       // while循环开始
    {   scan=1;                         // 初始扫描信号
        for(j=0;j<4;j++)                // for叙述开始
        {   P0=0xff;                    // 关7段显示器
            P1=~scan;                   // 输出扫描信号
            P0=TAB[disp[j]];            // 转换成驱动信号，并输出到P0
            delay1ms(4);                // 延迟4ms
            scan<<=1;                   // 下一个扫描信号
        }                               // 结束for叙述
    }                                   // 结束while叙述
}                                       // display函数结束
//====延迟函数====
void delay1ms(char x)
{   int i,j;                            // 声明变量
    for(i=0;i<x;i++)                    // 外循环
        for(j=0;j<120;j++);             // 内循环
}                                       // 延迟函数结束
```

ADC连续转换实例演练（ch11-1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图11-41连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。调整其中10kΩ可变电阻，看看7段LED数码管上有无变化？若没有变化，请检查线路有无错误？程序有无问题？并将它记录在实验报告里。

4. 若7段LED数码管上所指示的值会随可变电阻的调整而变动，再利用数字电表测量

“$+V_{in}$”引脚对地电压，并与7段LED数码管上所指示的值比较，以找出误差，并记录在实验报告里。

5. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

6. 编写实验报告。

11.7.2 ADC交互式转换实例演练之一

● 实验要点

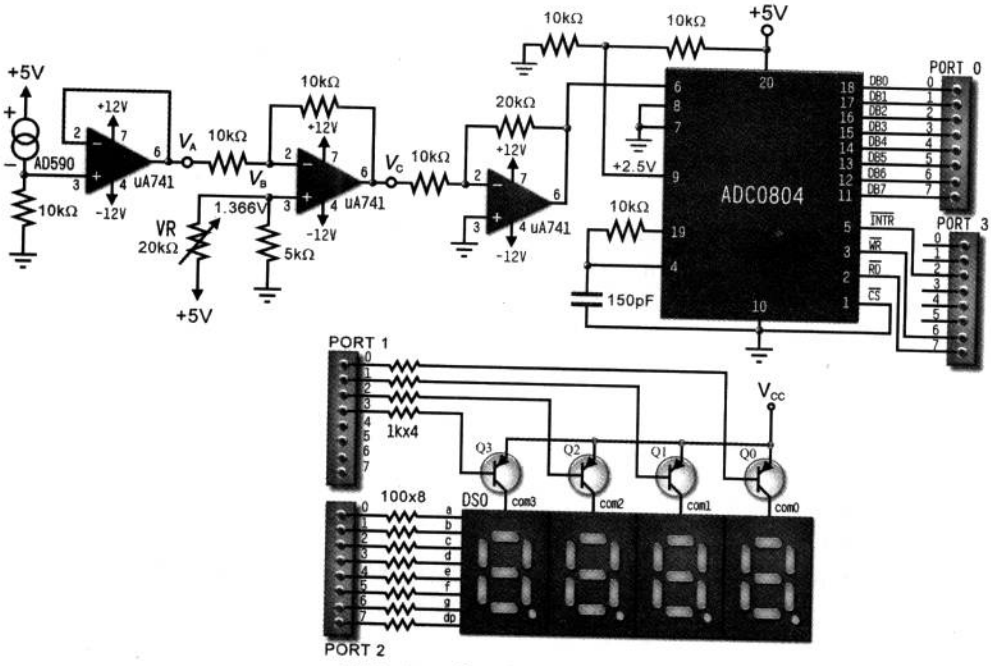


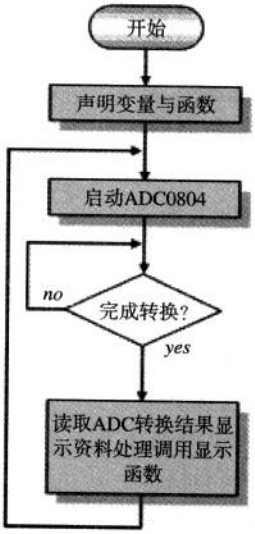
图11-42 交互式传输电路图

如图11-42所示，ADC0804连接到8051的PORT 0，而ADC0804的 $\overline{WR}$ 引脚连接到8051的 $\overline{WR}$ 引脚，ADC0804的 $\overline{RD}$ 引脚连接到8051的 $\overline{RD}$ 引脚，让ADC0804变成是8051的“外部存储器”，而ADC0804的 $\overline{INTR}$ 引脚连接到8051的P3.2引脚，可当成一般输入口，以查询方式检测ADC0804是否完成转换；也可以中断方式处理。将转换后的数字信号，输出到PORT 2所连接的4数字7段LED数码管，而其扫描信号是通过Port 1送出去的。

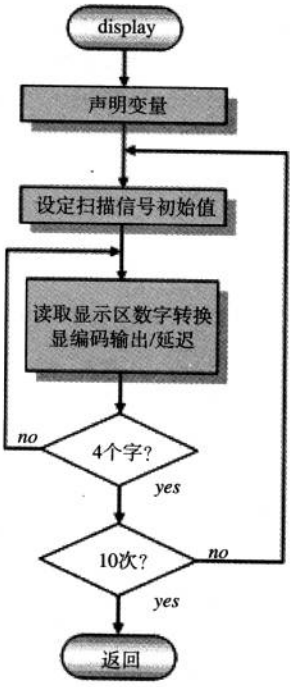
● 流程图与程序设计

根据功能需求与电路结构得知，当执行“储存外部存储器”时，8051将通过 $\overline{WR}$ 引脚通知ADC0804开始进行转换。紧接着，程序只要等待ADC0804由 $\overline{INTR}$ 引脚送来的完成转换信号，即可进行外部数据的读取。再将读取到的数据，进行数据处理后，送到

PORT 1即可显示该数值。



主程序




```
/* ADC交互式转换实例演练之一(ch11-2.c) */
#include <reg51.h>
/*声明驱动信号数组*/
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,
               0x92, 0x83, 0xf8, 0x80, 0x98 };
//=========================================
unsigned char disp[4]={0, 0, 0, 0};      // 声明显示区数组
unsigned char xdata  adc;                // 声明xdata存储器形式变量
sbit  INTR=P3^2;                         // 声明INTR变量
void display(void);                      // 声明显示函数
void delay1ms(char);                     // 声明延迟函数
main()                                   // 主程序
{    int results;                        // 声明变量
     while(1)                            // while开始
     {   adc=0xff;                       // 执行写入外部存储器的操作
                                         // ADC0804开始进行转换
         if(INTR==1)                     // 判断是否完成转换
         {   results=adc;                // 读取转换结果
```

```
            disp[3]=results/10000;          // 取得千位数
            disp[2]=(results/1000)%10;      // 取得百位数
            disp[1]=(results/100)%10;       // 取得十位数
            disp[0]=(results/10)%10;        // 取得个位数
        }
        display();                          // 调用显示函数
    }                                       // while结束
}                                           // 主程序结束
//====显示函数====
void display(void)
{   char j,scan;                            // 声明变量
    char i=10;                              // 扫描10次
    while(--i>=0)                           // if循环开始
    {   scan=1;                             // 初始扫描信号
        for(j=0;j<4;j++)                    // for叙述开始
        {   P2=0xff;                        // 关7段显示器
            P1=~scan;                       // 输出扫描信号
            P2=TAB[disp[j]];                // 转换成驱动信号，并输出到P2
            delay1ms(4);                    // 延迟4ms
            scan<<=1;                       // 下一个扫描信号
        }                                   // 结束for叙述
    }                                       // 结束if叙述
}                                           // display函数结束
//====延迟函数====
void delay1ms(char x)
{   char i,j;                               // 声明变量
    for(i=0;i<x;i++)                        // 外循环
        for(j=0;j<120;j++);                 // 内循环
}                                           // 延迟函数结束
```

ADC交互式转换实例演练之一（ch11-2.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图11-42连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。7段LED数码管显示为何？若将加热后的烙铁靠近AD590，7段LED数码管有无反应？若无反应或出现非预期的状况，则检查线路的连接

状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

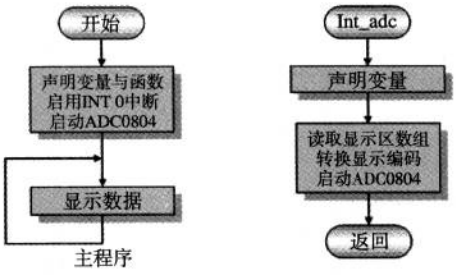
5. 编写实验报告。

11.7.3 ADC交互式转换实例演练之二

实验要点

本实验的功能与11.7.1节完全相同，不同的是，在本实验里将采用中断方式。

流程图与程序设计



首先声明变量与函数，再设定中断，然后执行“储存外部存储器”时，以通知ADC0804开始进行转换，而主程序就持续执行扫描显示。在中断子程序里，首先读取外部数据的读取，将它进行显示数据处理后，然后放入显示区数组，重新启动ADC0804即可。

```
/* ADC交互式转换实例演练之二(ch11-3.c) */
#include <reg51.h>
/*声明驱动信号数组*/
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,
               0x92, 0x83, 0xf8, 0x80, 0x98 };
//=========================================
unsigned char disp[4]={0, 0, 0, 0};          // 声明显示区数组
unsigned char xdata  adc;                    // 声明xdata存储器形式变量
unsigned char results;                       // 声明变量
void delay1ms(char);                         // 声明延迟函数
//====主程序====
main()                                       // 主程序
{    char i,scan;                            // 声明变量
     IE=0x81;                                // 启用INT0中断
     TCON=0x01;                              // 采用负边缘触发
```

```
    adc=0xff;                                    // 采用启动ADC0804
    while(1)                                     // while开始
    {   scan=1;                                  // 初始扫描信号
        for(i=0;i<4;i++)                         // for叙述开始
        {   P2=0xff;                             // 关7段显示器
            P1=~scan;                            // 输出扫描信号
            P2=TAB[disp[i]];                     // 转换成驱动信号，并输出到P2
            delay1ms(4);                         // 延迟4ms
            scan<<=1;                            // 下一个扫描信号
        }                                        // 结束for叙述
    }                                            // 结束while叙述
}
//====中断函数====
void int_adc(void)interrupt 0
{   results= adc;                                // 读取转换结果
    disp[3]=results/10000;                       // 取得千位数
    disp[2]=(results/1000)%10;                   // 取得百位数
    disp[1]=(results/100)%10;                    // 取得十位数
    disp[0]=(results/10)%10;                     // 取得个位数
    adc=0xff;                                    // 采用启动ADC0804
}                                                // 中断程序结束
//====延迟函数====
void delay1ms(char x)
{   char i,j;                                    // 声明变量
    for(i=0;i<x;i++)                             // 外循环
        for(j=0;j<120;j++);                      // 内循环
}                                                // 延迟函数结束
```

ADC交互式转换实例演练之二（ch11-3.c）

● 操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图11-28连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。7段LED数码管显示为何？若将加热后的烙铁靠进AD590，7段LED数码管有无反应？若无反应或出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实

体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

11.7.4 ADC的温控实例演练

实验要点

在本单元里所使用的电路与前面的单元类似，只是多出一个继电器的控制，如图11-43所示，由P3.3输出到晶体管，以驱动继电器。当温度超过35℃时，关闭继电器；当温度低于20℃时，启动继电器。

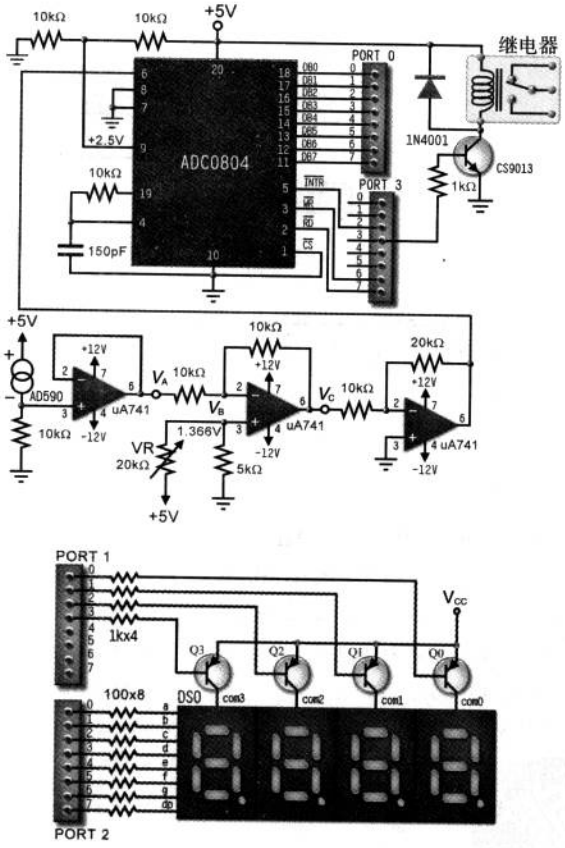
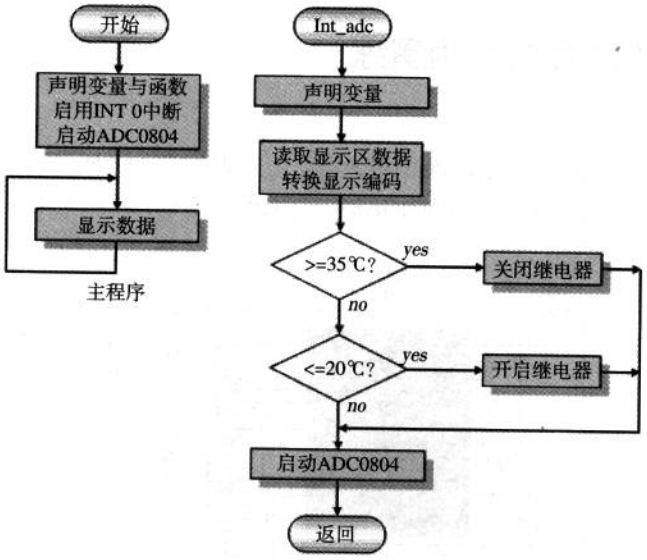


图11-43 温控实验电路图

流程图与程序设计

根据功能需求与电路结构得知，首先由8051送一个开始转换的信号给ADC0804，以启动ADC0804，再等待ADC0804传回已完成转换的信号，即可读入温度数据。若继电器on，则判断温度是否达到上限，若是则关闭继电器，然后显示温度。若继电器on，而温度未达上限，则直接显示温度。若继电器off，则判断温度是否达到下限，若是则开启继

电器，然后显示温度、若继电器off，而温度未达下限，则直接显示温度。显示温度之后，再重新开始。




```
/* ADC的温控实验(ch11-4.c) */
#include <reg51.h>
#define off  35
#define on  20
/*声明驱动信号数组*/
char TAB[10]={ 0xc0, 0xf9, 0xa4, 0xb0, 0x99,
               0x92, 0x83, 0xf8, 0x80, 0x98 };
//=========================================
unsigned char disp[4]={0, 0, 0, 0};        // 声明显示区数组
unsigned char xdata  adc;                  // 声明xdata存储器形式变量
sbit relay = P3^3;                         // 声明继电器位置
unsigned char results;                     // 声明变量
void delay1ms(char);                       // 声明延迟函数
//====主程序====
main()                                     // 主程序
{    char i, scan;                         // 声明变量
     IE=0x81;                              // 启用INT0中断
     TCON=0x01;                            // 采用负边缘触发
     adc=0xff;                             // 采用启动ADC0804
     relay=0;                              // 关闭继电器
```

```
    while(1)                          // while开始
    {   scan=1;                       // 初始扫描信号
        for(i=0;i<4;i++)              // for叙述开始
        {   P2=0xff;                  // 关7段显示器
            P1=~scan;                 // 输出扫描信号
            P2=TAB[disp[i]];          // 转换成驱动信号，并输出到P2
            delay1ms(4);              // 延迟4ms
            scan<<=1;                 // 下一个扫描信号
        }                             // 结束for叙述
    }                                 // 结束while叙述
}
//====中断函数====
void int_adc(void) interrupt 0
{   results= adc;                     // 读取转换结果
    disp[3]=results/10000;            // 取得千位数
    disp[2]=(results/1000)%10;        // 取得百位数
    disp[1]=(results/100)%10;         // 取得十位数
    disp[0]=(results/100)%10;         // 取得个位数
    if(results >=off)relay=0;
    /*若温度高于或等于35℃，则关闭继电器*/
    else if(results <=on)relay=1;
    /*若温度低于或等于20℃，则开启继电器*/
    adc=0xff;                         // 采用启动ADC0804
}                                     // 中断程序结束
//====延迟函数====
void delay1ms(char x)
{   char i,j;                         // 声明变量
    for(i=0;i<x;i++)                  // 外循环
        for(j=0;j<120;j++);           // 内循环
}                                     // 延迟函数结束
```

ADC的温控实验（ch11-4.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图11-43连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若有出现预期的状况，则检查线路的连接状况，

看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，看看是否正常？

5. 编写实验报告。

11.7.5 DAC实例演练

实验要点

如图11-44所示，指拨开关的状态由P0输入，而开关状态经P2连接到DAC-08，将它转换成模拟电压。请以数字电表测量电路中的V_o，观察其输出的模拟电压，是否符合指拨开关的状态？

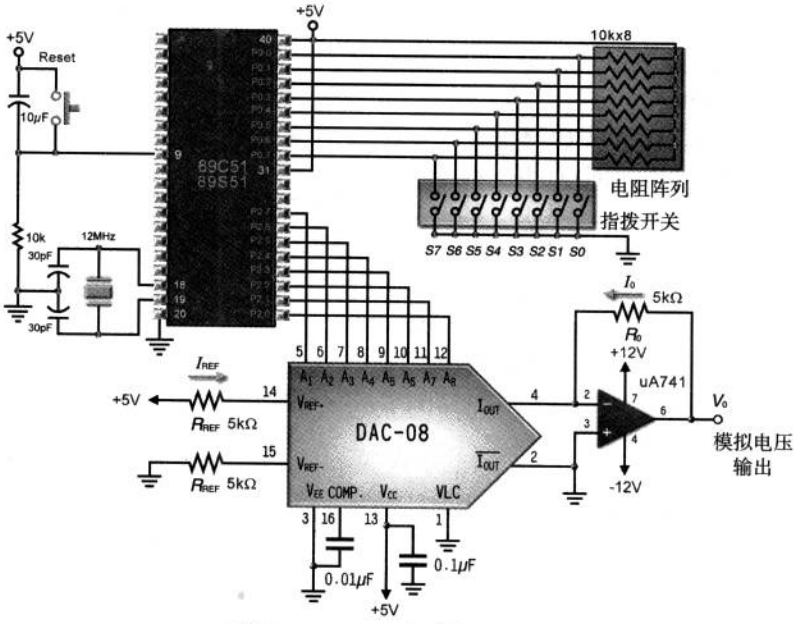


图11-44 DAC实验电路图

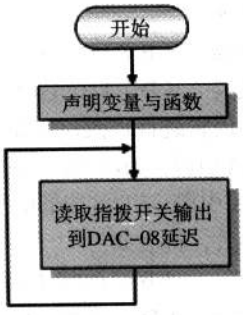
流程图与程序设计

根据功能需求与电路结构得知，首先将P0设计成输入口，再读取指拨开关的状态，然后将它输出到P2，让DAC-08将它转换成模拟电压。经过一段时间延迟（任意时间）后，再重复进行上述操作。

```
/* DAC实验(ch11-5.c) */
#include <reg51.h>
void delay1ms(char);                    // 声明延迟函数
main()                                  // 主程序
```

```
{    P0=0xff;                          //设计输入
     while(1)                          //while开始
     {                                 //读取指拨开关
         P2=~P0;                       //输出到DAC
         delay1ms(1);                  //调用延迟函数
     }                                 //while结束
}                                      //主程序结束
//====延迟函数====
void delay1ms(char x)
{    char i,j;                         //声明变量
     for(i=0;i<x;j++)                  //外循环
         for(j=0;j<120;j++);           //内循环
}                                      //延迟函数结束
```

DAC实验（ch11-5.c）



主程序

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图11-44连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，再以数字电表测量V_O的电压，记录每个指拨开关状态下的V_O，并记录在实验报告里。

5. 编写实验报告。

11.8 实时练习



在本章里探讨了ADC的原理与实用IC、DAC的原理与实用IC，还有简易的温控IC，可说是8051应用系统中相当重要的一部分。在此请试着回答下列问题，以确认对于此部分的掌握程度。

选择题

（　）1. 下列哪种AD转换器的转换速度比较快？
（A）双斜率型AD转换器　（B）比较型AD转换器
（C）连续计数式AD转换器　（D）连续逼近式AD转换器

（　）2. 下列哪种AD转换器的精密度比较高？
（A）双斜率型AD转换器　（B）比较型AD转换器
（C）连续计数式AD转换器　（D）连续逼近式AD转换器

（　）3. ADC0804具有什么功能？
（A）8位模拟-数字转换器　（B）11位模拟-数字转换器
（C）8位数字-模拟转换器　（D）11位数字-模拟转换器

（　）4. 若要启动ADC0804，使它进行转换，应如何处理？
（A）加高电平信号到 $\overline{CS}$ 引脚　（B）加高电平信号到 $\overline{WR}$ 引脚
（C）加低电平信号到 $\overline{CS}$ 引脚　（D）加低电平信号到 $\overline{WR}$ 引脚

（　）5. 当ADC0804完成转换后，将会如何？
（A）$\overline{CS}$ 引脚转为低电平　（B）$\overline{CS}$ 引脚转为高电平
（C）$\overline{INTR}$ 引脚转为低电平　（D）$\overline{INTR}$ 引脚转为高电平

（　）6. 下列哪个IC具有温度感应功能？
（A）DAC-08　（B）AD590　（C）uA741　（D）NE555

（　）7. 下列哪种数为信号转换模拟信号的方式比较实际？
（A）R-2R电阻网络　（B）加权电阻网络
（C）双Y形电阻网络　（D）三角形电阻网络

（　）8. 当温度每上升1℃时，AD590会有什么变化？
（A）电压上升1mV　（B）电压下降1mV
（C）电流上升1μA　（D）电流下降1μA

（　）9. 若要让ADC0804进行连续转换，应如何连接？
（A）$\overline{CS}$ 引脚与 $\overline{INTR}$ 引脚连接、$\overline{WR}$ 引脚与 $\overline{RD}$ 引脚接地
（B）$\overline{CS}$ 引脚与 $\overline{WR}$ 引脚连接、$\overline{INTR}$ 引脚与 $\overline{RD}$ 引脚接地
（C）$\overline{WR}$ 引脚与 $\overline{INTR}$ 引脚连接、$\overline{CS}$ 引脚与 $\overline{RD}$ 引脚接地
（D）$\overline{RD}$ 引脚与 $\overline{INTR}$ 引脚连接、$\overline{WR}$ 引脚与 $\overline{CS}$ 引脚接地

（　）10. 若要ADC080与8051采用交互式信号传输，则应怎么做？
（A）8051将ADC0804视为外部存储器

（B）8051通过Port 0连接ADC0804的数据总线

（C）8051的 $\overline{RD}$ 引脚与ADC0804的 $\overline{RD}$ 引脚相连接、8051的 $\overline{WR}$ 引脚与ADC0804的 $\overline{WR}$ 引脚相连接

（D）以上皆是

问答题

1. 试述模拟信号与数字信号的特性。

2. 简述并行式ADC的原理及其特性。

3. 简述连续逼近式ADC的原理及其特性。

4. 简述连续计数式ADC的原理及其特性。

5. 简述双斜率式ADC的原理及其特性。

6. 试述ADC0804的特性。

7. 试述ADC0804所能接受的时钟脉冲频率范围，如何利用其内部振荡电路产生时钟脉冲。

8. 若要将ADC0804视为外部存储器，在C语言程序里，应如何声明？而ADC0804的 $\overline{WR}$ 引脚、$\overline{RD}$ 引脚与 $\overline{INTR}$ 引脚应如何连接。

9. 试述AD590的用途与特性。

10. 试设计一个AD590与ADC0804的接口，使温度变化1℃，在ADC0804的输出数字信号就增减1。

（B）8051单片机0号与ADC0804的接线方式

（C）8051的P0口引脚与ADC0804的DB引脚相连接，8051的WR引脚与ADC0804的WR引脚相连接

（D）正确答案

1. 试比较两种ADC的优缺点。

2. 简述ADC0804的工作过程。

3. 简述单片机与ADC0804的接口方法。

4. 简述逐次逼近式ADC的原理及特点。

5. 简述双积分式ADC的原理及特点。

6. 比较ADC0804和ADC0809的异同。

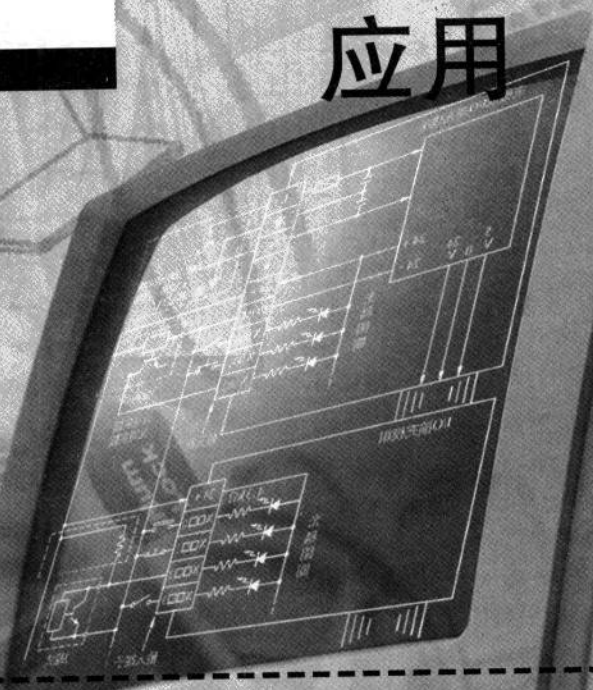
7. 在ADC0804的典型应用电路中，如果输入电压为0～5V，则输出电压与输入电压的关系是什么？

8. 若将ADC0804改为外部时钟源，该如何调整？而ADC0804的WR引脚、RD引脚与INTR引脚应如何连接？

9. 试比较ADC0809与ADC0804的异同。

10. 简述什么是ADC0809的接口，通道选择IN4，在ADC0804的测出数字量后如何调用？

第12章 LED阵列的应用



本章内容丰富，主要包括以下两部分。

硬件部分：

LED阵列的结构与现有的LED阵列器件。

LED阵列的驱动电路与动态文字图案的演示方式。

程序与实践部分：

8×8LED阵列的静态演示程序，动态演示程序，以及16×16 LED阵列的驱动程序。

12.1 认识LED阵列



所谓LED阵列是将多个LED以矩阵方式排列成为一个器件，其中各LED的引脚有规律地连接，如图12-1左图所示为共阳型5×7 LED阵列内部电路结构。

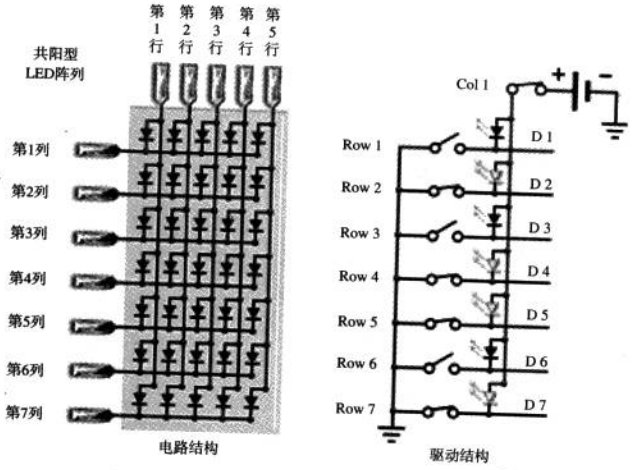


图12-1 共阳极型5×7 LED阵列结构

对于共阳型5×7 LED阵列而言，每行LED的阳极连接在一起，即为行引脚（column）；而每列LED的阴极连接在一起，即为列引脚（row）。通常是站在行的角度来看，所以称为“共阳型”（common anode，简称**CA**）。若要点亮其中的LED，则需要行的信号与列的信号交集。例如要第1行、第2列的LED（即D2）亮，则需要将Col. 1引脚接到电源（VCC），Row 2引脚接地，才能形成一个正向回路，该LED才会亮，如图12-1右图所示。送到行引脚的信号为扫描信号，在图12-1所示的5个行信号之中，只有一个为高电平，其余为低电平，称为高电平扫描。换言之，任何时间里只有一行的LED可能会亮。而所要点亮的信号则由列引脚送入低电平信号。当信号切换的速度够快时，我们将感觉到整个LED阵列是亮的，而不是只亮其中一行而已。

若连接到行引脚的是LED的阴极，则称为“共阴型”（common cathode，简称**CC**）。若要点亮这种LED阵列，其行引脚必须采用低电平扫描，而列引脚为高电平信号。表12-1所示为常见的市售LED阵列。

表12-1

编号	尺寸	型式	颜色	包装	编号	尺寸	型式	颜色	包装
MM07574P	0.7",5×7	共阳	红	D-38	MM07573P	0.7",5×7	共阴	红	D-38
MM07574G	0.7",5×7	共阳	绿	D-38	MM07573G	0.7",5×7	共阴	绿	D-38
MM07574A	0.7",5×7	共阳	高亮度红或橙	D-38	MM07573A	0.7",5×7	共阴	高亮度红或橙	D-38
MM12574P	1.2",5×7	共阳	红	D-39	MM12573P	1.2",5×7	共阴	红	D-39
MM12574G	1.2",5×7	共阳	绿	D-39	MM12573G	1.2",5×7	共阴	绿	D-39

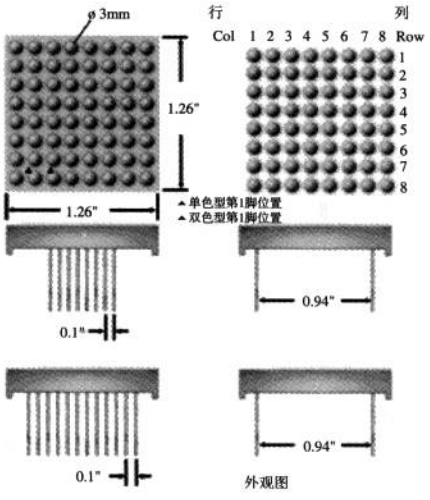
续表

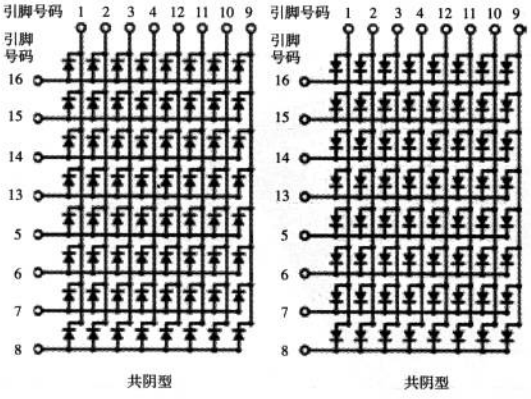
编号	尺寸	型式	颜色	包装	编号	尺寸	型式	颜色	包装
MM12574A	1.2",5×7	共阳	高亮度红或橙	D-39	MM12573A	1.2",5×7	共阴	高亮度红或橙	D-39
MM20574P	2.0",5×7	共阳	红	D-40	MM20573P	2.0",5×7	共阴	红	D-40
MM20574A	2.0",5×7	共阳	橙	D-40	MM20573A	2.0",5×7	共阴	橙	D-40
MM20574AG	2.0",5×7	共阳	橙/绿	D-40	MM20573AG	2.0",5×7	共阴	橙/绿	D-40
MM14584P	1.4",5×8	共阳	红	D-41	MM14583P	1.4",5×7	共阴	红	D-41
MM14584G	1.4",5×8	共阳	绿	D-41	MM14583G	1.4",5×7	共阴	绿	D-41
MM14584A	1.4",5×8	共阳	高亮度红或橙	D-41	MM14583A	1.4",5×7	共阴	高亮度红或橙	D-41
MM23584P	2.3",5×8	共阳	红	D-42	MM23583P	2.3",5×8	共阴	红	D-42
MM23584A	2.3",5×8	共阳	橙	D-42	MM23583A	2.3",5×8	共阴	橙	D-42
MM23584AG	2.3",5×8	共阳	橙/绿	D-42	MM23583AG	2.3",5×8	共阴	橙/绿	D-42
MM12884P	1.2",8×8	共阳	红	D-43	MM12883P	1.2",8×8	共阴	红	D-43
MM12884A	1.2",8×8	共阳	橙	D-43	MM12883A	1.2",8×8	共阴	橙	D-43
MM12884AG	1.2",8×8	共阳	橙/绿	D-43	MM12883AG	1.2",8×8	共阴	橙/绿	D-43
MM23884P	2.3",8×8	共阳	红	D-44	MM23883P	2.3",8×8	共阴	红	D-44
MM23884A	2.3",8×8	共阳	橙	D-44	MM23883A	2.3",8×8	共阴	橙	D-44
MM23884AG	2.3",8×8	共阳	橙/绿	D-44	MM23883AG	2.3",8×8	共阴	橙/绿	D-44

上表所示为常用的5×7LED、5×8LED及8×8 LED阵列，所谓5×7 LED阵列就是由5行（横向）、7列（纵向）LED所组成的阵列，同理，5×8 LED阵列就是5行、8列LED，8×8 LED阵列就是8行、8列LED。由于LED阵列的种类繁多，在此仅介绍常用的8×8 LED阵列。

● 8×8 LED阵列

根据尺寸区分，在8×8 LED阵列可分为1.2英寸及2.3英寸两种，1.2英寸采用D-43包装，其外观、结构与引脚，如图12-2所示；2.3英寸采用D-44包装其外观、结构与引脚，如图12-3所示。根据连接方式区分，则有共阳与共阴两类。根据颜色区分，则有红、绿、亮红或橙色以及双色LED（MM12884AG、MM12883AG、MM23884AG及MM23883AG）。



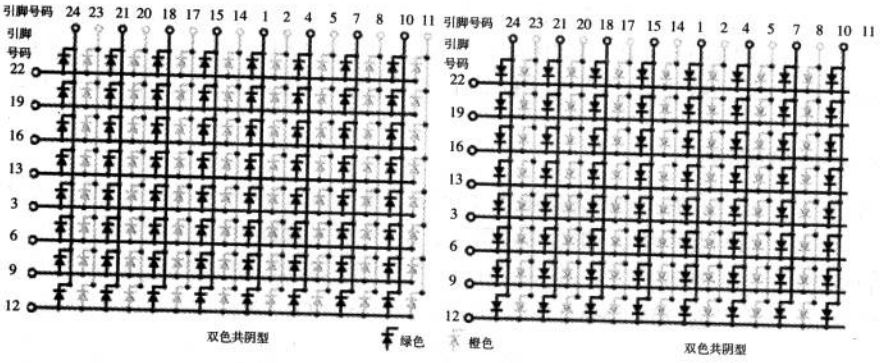


引脚号码	端点	引脚号码	端点	引脚号码	端点
1	Col. 1	7	Row 7	13	Row 4
2	Col. 2	8	Row 8	14	Row 3
3	Col. 3	9	Col. 8	15	Row 2
4	Col. 4	10	Col. 7	16	Row 1
5	Row 5	11	Col. 6		
6	Row 6	12	Col. 5		

单色引脚表（根据器件的DATA SHEET）

引脚号码	端点	引脚号码	端点	引脚号码	端点
1	Row 5	7	Row 6	13	Col. 1
2	Row 7	8	Row 3	14	Row 2
3	Col. 2	9	Row 1	15	Col. 7
4	Col. 3	10	Col. 4	16	Col. 8
5	Row 8	11	Col. 6		
6	Col. 5	12	Row 4		

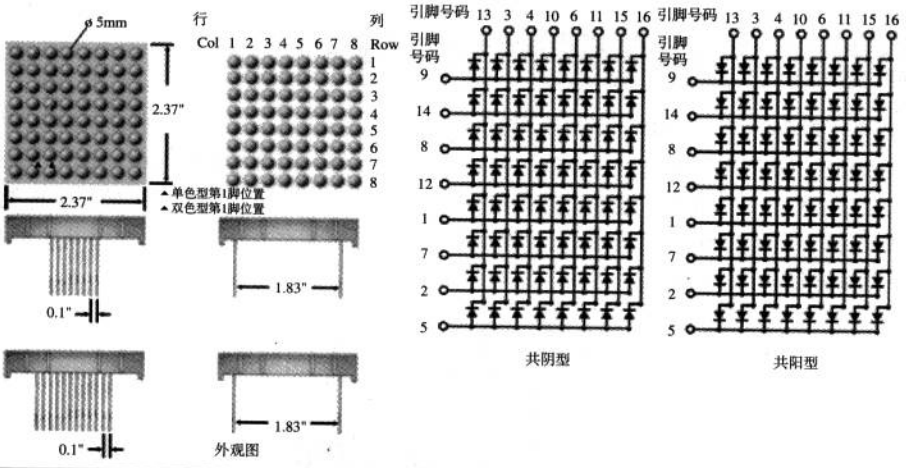
单色引脚表（根据市售的器件，与D-44相同）



引脚号码	端点	引脚号码	端点	引脚号码	端点
1	Col. 5（绿）	9	Row 7	17	Col. 3（橙）
2	Col. 5（橙）	10	Col. 8（绿）	18	Col. 3（绿）
3	Row 5	11	Col. 8（橙）	19	Row 2
4	Col. 6（绿）	12	Row 8	20	Col. 2（橙）
5	Col. 6（橙）	13	Row 4	21	Col. 2（绿）
6	Row 6	14	Col. 4（橙）	22	Row 1
7	Col. 7（绿）	15	Col. 4（绿）	23	Col. 1（橙）
8	Col. 7（橙）	16	Row 3	24	Col. 1（绿）

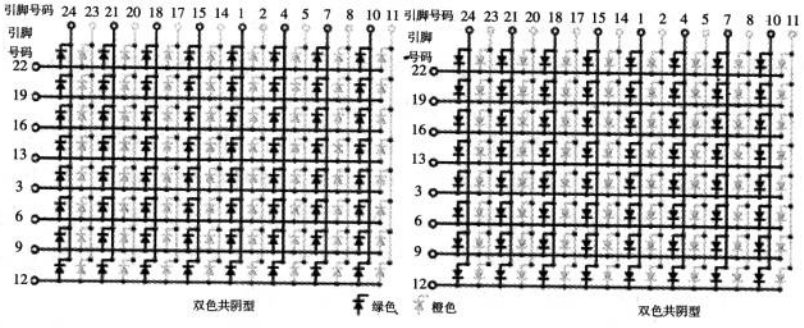
双色引脚表

图12-2　1.2英寸包装（D-43）



引脚号码	端点	引脚号码	端点	引脚号码	端点
1	Row 5	7	Row 6	13	Col. 1
2	Row 7	8	Row 3	14	Row 2
3	Col. 2	9	Row 1	15	Col. 7
4	Col. 3	10	Col. 4	16	Col. 8
5	Row 8	11	Col. 6		
6	Col. 5	12	Row 4		

单色引脚表



引脚号码	端点	引脚号码	端点	引脚号码	端点
1	Col. 5（绿）	9	Row 7	17	Col. 3（橙）
2	Col. 5（橙）	10	Col. 8（绿）	18	Col. 3（绿）
3	Row 5	11	Col. 8（橙）	19	Row 2
4	Col. 6（绿）	12	Row 8	20	Col. 2（橙）
5	Col. 6（橙）	13	Row 4	21	Col. 2（绿）
6	Row 6	14	Col. 4（橙）	22	Row 1
7	Col. 7（绿）	15	Col. 4（绿）	23	Col. 1（橙）
8	Col. 7（橙）	16	Row 3	24	Col. 1（绿）

双色引脚表

图12-3 2.3英寸包装（D-44）

12.2 LED阵列驱动电路



若要正向点亮一片LED，至少也得10~20mA；如果电流不够大，则LED不够亮！而不管是8051的输入/输出口，还是TTL、CMOS的输出端，其高电平输出电流都不是很高，大不了1~2mA而已！因此，很难直接高电平驱动LED。这时候就需要额外的驱动电路了，LED阵列的驱动电路包括两组信号，即扫描信号和显示信号。下面分别介绍共阳型与共阴型LED阵列的驱动电路。

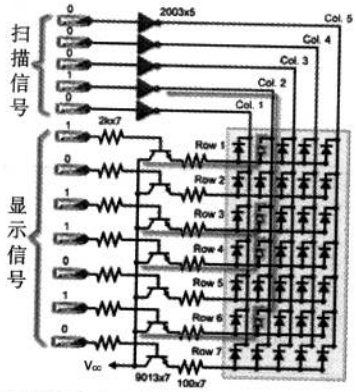


图12-4 共阴型高电平扫描高电平显示信号驱动电路

共阴型高电平扫描、高电平显示信号驱动电路

图12-4所示为针对共阴型LED阵列设计的驱动电路，这种驱动电路采用高电平扫描，也就是任何时间只有一个高电平信号，其他则为低电平。一行扫描完成后，再把高电平信号转到邻近的其他行。扫描信号接用一个反相驱动器，如ULN2003之类的IC，其输出为低电平，连接LED阵列的行引脚。这种反相驱动器属于集电极式输出，输出低电平时，最大可吸取0.5A，即500mA，就算每个LED取用30mA，7个LED同时亮，也只不过210mA，使用ULN2003即可。当高电平的扫描信号输入ULN2003后，将输出为低电平，连接该行LED的阴极，即可使该行中的LED具有点亮的条件。

在此，显示信号各经过一个2kΩ限流电阻送入晶体管的基极，而每个NPN晶体管的集电极连接VCC，射极输出经一个100Ω的限流电阻连接到LED阵列的列引脚。对于高电平的显示信号，将可提供其所连接LED的驱动电流，而这个驱动电流经过LED到ULN2003的输出端，形成正向回路，即可点亮该LED。其中每个晶体管任何时间，只需负责驱动一个LED，所以只要找一个能够提供30mA射极电流的晶体管即可，采用学校里广泛使用的CS9013就可以了！

在图12-4中扫描信号为“01000”，在Col. 2行里的所有LED的阴极为低电平；显示信号为“1011010”可能会亮，所以Row 1、Row 3、Row 4、Row 6为高电平。两组信号交集，所以Col. 2行里的Row 1、Row 3、Row 4、Row 6等4片LED正向导通而亮。

● 共阴型低电平扫描、高电平显示信号驱动电路

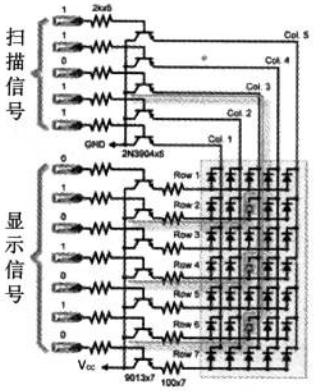


图12-5 共阴型低电平扫描高电平显示信号驱动电路

如图12-5所示为共阴型LED阵列低电平扫描信号、高电平显示信号驱动电路。这种扫描方式是任何时间只有一个低电平信号，其他则为高电平。一行扫描完成后，再把低电平信号转到邻近的其他行。扫描信号经限流电阻，连接于PNP晶体管的基极。晶体管的集电极接地，射极则连至LED阵列的行引脚。若要同时点亮该行的7个LED，则该晶体管的射极电流必须大于或等于210mA才行！常用的CS9015、2N3904A、2SA684、2N4037等都可以达得到。当低电平的扫描信号输入晶体管的基极后，该晶体管即为正向，而产生I_B、I_E，连接该行LED的阴极，即可使该行中的LED具有点亮的条件。

所要显示的信号各经过一个限流电阻送入晶体管的基极，而每个NPN晶体管的集电极连接V_{CC}，射极输出经一个100Ω的限流电阻连接到LED阵列的列引脚。对于高电平的显示信号，将可提供其所连接LED的驱动电流，而这个驱动电流经过LED到其所连接晶体管（2N3904）的射极，形成正向回路，即可点亮该LED。同样地，其中每个晶体管任何时间，只需负责驱动一个LED，所以只要找一个能够提供30mA射极电流的晶体管即可，采用学校里广泛使用的CS9013就可以了！

图12-5中的扫描信号为“11011”，在Col. 3行里的所有LED的阴极为低电平；显示信号为“0101010”可能会亮，所以Row 2、Row 4和Row 6为高电平。两组信号交集，所以Col. 3行里的Row 2、Row 4和Row 6三片LED正向导通点亮。

● 共阳型高电平扫描高电平显示信号驱动电路

如图12-6所示为针对共阳型LED阵列而设计的驱动电路，这种驱动电路采用高电平扫描，也就是任何时间只有一个高电平信号，其他为低电平。一行扫描完成后，再把高电平信号转到邻近的其他行。扫描信号连接到一个NPN晶体管的基极，同样这个晶体管必须能够提供7个LED同时亮，所需要的电流约210mA。在此选用常用的CS9013、2SC1384、2N3053等即可。当高电平的扫描信号输入后，晶体管的射极电流将流入行LED的阳极，即可满足该行中的LED点亮的条件。

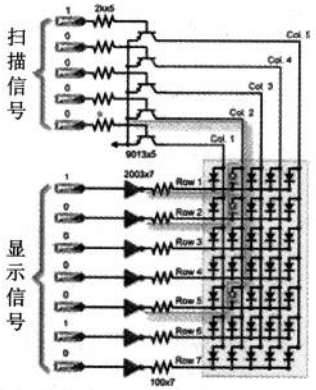


图12-6　共阳型高电平扫描高电平显示信号驱动电路

所要显示的信号各经过一个反相驱动器（如ULN2003），再经限流电阻接到LED阵列的列引脚。对于高电平的显示信号，经反相驱动器后，即为低电平，即可吸取所连接LED的驱动电流，而形成正向回路点亮该LED。虽然其中每个反相驱动器任何时间，只须负责驱动一个LED，对于ULN2003而言，有点大材小用，使用这个IC比使用7个晶体管还简单！

在图12-6中扫描信号为“10000”，在Col. 5行里的所有LED的阴极为高电平；显示信号为“1000010”可能会亮，所以Row 1、Row 6为低电平。两组信号交集，所以Col. 5行里的Row 1、Row 6等两片LED正向导通而亮。

● 共阳型低电平扫描高电平显示信号驱动电路

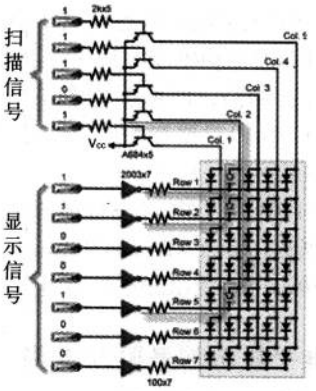


图12-7　共阳型低电平扫描高电平显示信号驱动电路

如图12-7所示是针对共阳型LED阵列而设计的驱动电路，这种驱动电路采用低电平扫描，也就是在任何时间内只有一个低电平信号，其他则为高电平。一行扫描完成后，再把低电平信号转到邻近的其他行。扫描信号连接到一个PNP晶体管的基极，而该晶体管的射极接到V_{cc}。同样地，这个晶体管必须能够提供7个LED同时亮，所需要的电流约为210mA。当低电平的扫描信号输入后，晶体管的集电极电流，将可流入行LED的阳极，

即可使该行中的LED具有点亮的条件。

显示信号的部分与共阳型高电平扫描高电平显示信号驱动电路一样，在此不赘述。图12-7中的扫描信号为“10111”，在Col. 2行里的所有LED的阴极为高电平；显示信号为“1100100”可能会亮，所以Row 1、Row 2和Row 5为低电平。两组信号交集，所以Col. 2行里的Row 1、Row 2和Row 5三片LED正向导通点亮。

若要并接多个LED阵列，例如使用4个16×16LED阵列，连接成为8×8LED阵列，则一个扫描信号同时驱动两行LED（8+8），如图12-8所示。

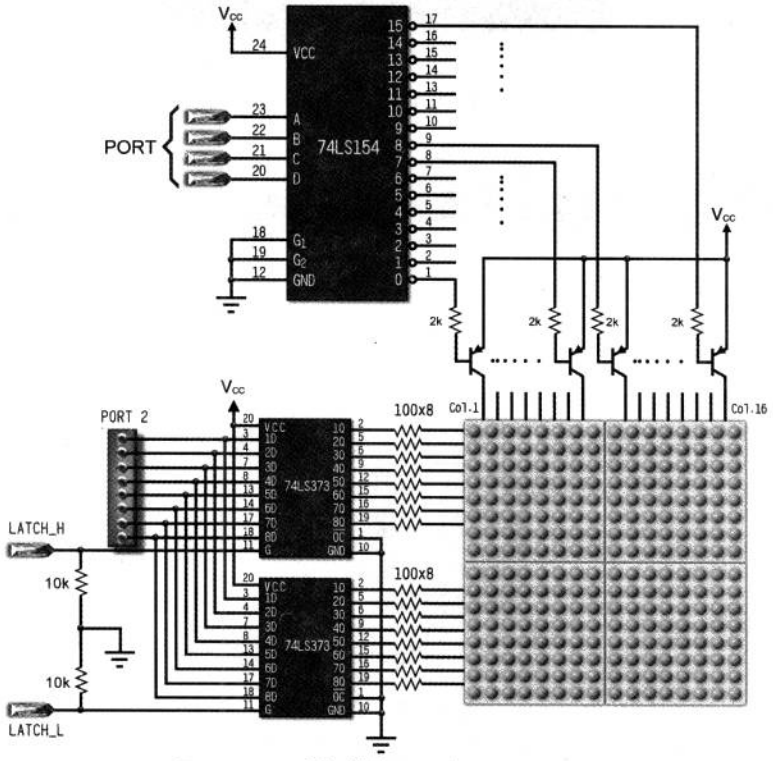


图12-8 一个扫描信号驱动两组显示信号

这时候就要靠锁存器（74LS373）将这两组显示信号锁住，在此的锁存器是以低电平输出的，其I_{OL}可达24mA，足以驱动一个LED；若还嫌不足，可改用**74AS373**（如图12-9所示），其I_{OL}最大为48mA。当74LS373的G脚为高电平时，数据可从输入端传输到锁存器里；G脚为低电平时，则数据将被锁住，不会随输入端而变。另外，$\overline{OC}$脚为输出控制引脚，当$\overline{OC}$脚为高电平时，输出端呈现高阻抗；$\overline{OC}$脚为低电平时，数据会从锁存器输出。

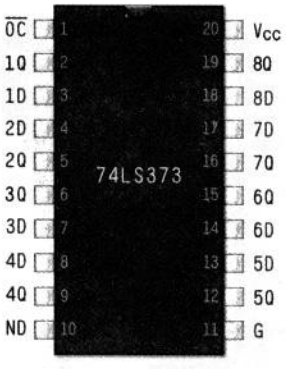


图12-9 74LS373

这个驱动电路的扫描信号总共16条，若直接由8051输出，将占用两个PORT，不太理想。在此使用一个4-16线译码器

（74LS154），这个译码器是将输入的16进制码译码后，输出低电平扫描信号。而输出低电平扫描信号经过限流电阻，连接到PNP晶体管的基极，再经过晶体管放大后即可驱动16个LED。

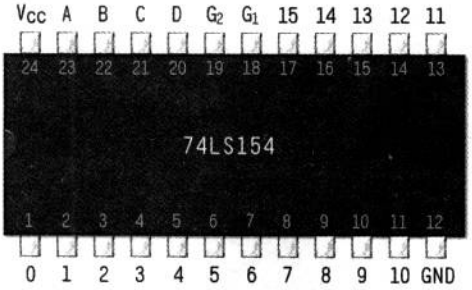


图12-10 74LS154

12.3 LED阵列显示方式



LED阵列的显示是采用扫描的方式，首先将所要显示的文字按每行拆解成多组显示信号。如下图所示，对于一个8×8 LED阵列而言，若要显示“公”，则可将各行显示数据如图12-11所示。

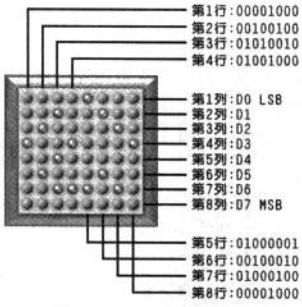


图12-11 文字编码

若LED阵列的第1列为显示数据的D0，第8列为D7，则这个字的显示数据编码如表12-2所示。

表12-2

扫描顺序	显示数据（2进制制）	显示数据（16进制制）
第1行	00001000	0x08
第2行	00100100	0x24
第3行	01010010	0x52
第4行	01001000	0x48
第5行	01000001	0x41
第6行	00100010	0x22
第7行	01000100	0x44
第8行	00001000	0x08

编码方式必须与实际线路相符，若第1列为LSB、第8列为MSB，则连接到微控制器时，也一定要按这样的顺序才行。当然，若要把第1列为MSB、第8列为LSB，则线路连接也要跟着调整。

LED阵列的显示方式就是按显示数据编码的顺序，一行一行地显示。以高电平扫描为例，若要显示第一行，则先将第一行的显示数据（00001000）送至LED阵列的列引脚，再将“10000000”扫描信号送至LED阵列的行引脚，即可显示第一行，此时其他行并不显示。同样地，若要显示第二行，则先将第二行的显示数据（00100100）送至LED阵列的列引脚，再将“01000000”扫描信号送至LED阵列的行引脚，即可显示第二行，此时其他行并不显示，依次类推，如图12-12所示。

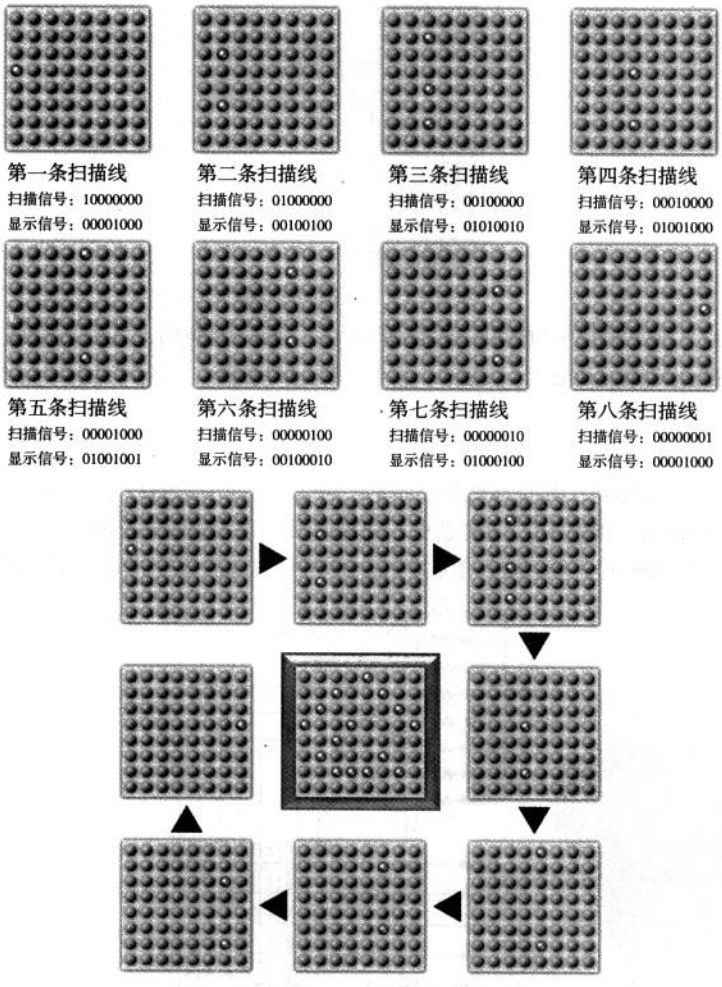


图12-12 数据扫描

每行的显示时间约4ms，由于人类的视觉暂留现象，将感觉到8行LED是在同时显示。若显示时间太短，则亮度不够；若显示时间太长，将会感觉到闪烁。

12.4 LED阵列动态显示



在LED阵列里，可以通过动态显示方式让所要显示的文字或图形左右移动或上下移动。在本单元里，将探讨如何水平移动和垂直移动。

12.4.1 水平移动

若要将文字或图形在LED阵列左右移动，只要以不同的顺序显示其编码即可，以如图12-13所示的编码为例。

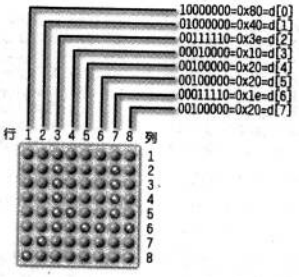


图12-13 “μ”的编码

若按第1行、第2行……的顺序重复循环显示，则LED阵列上将显示如图12-13所示的字样。下面将介绍如何让它有“左移”或“右移”的感觉。

左移

对于8×8 LED阵列，其左移就是显示8个不同的字型，如图12-14所示。首先扫描第一个字型，同样是8行、8次扫描、8次显示；完成第一个字型后，再扫描第二个字型；完成第二个字型后，再扫描第三个字型，依次类推，即可产生该文字字型或图形左移的感觉。

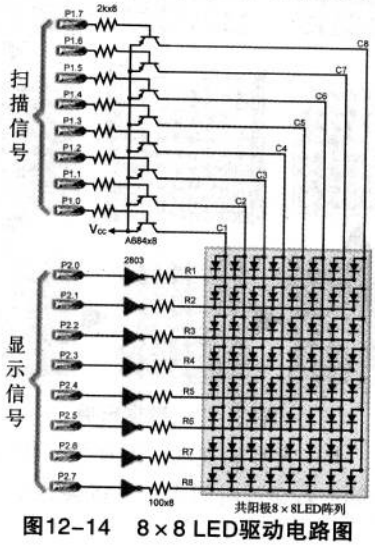


图12-14 8×8 LED驱动电路图

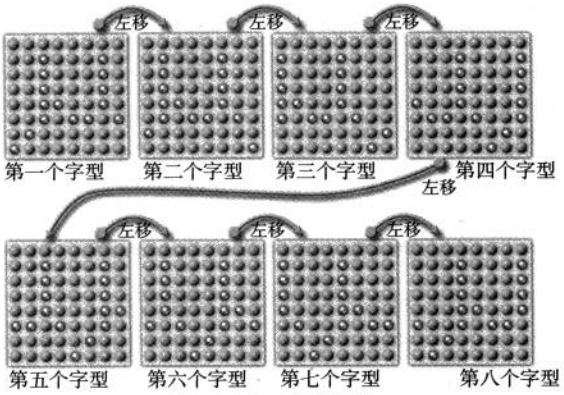


图12-15 左移步骤

原本的字型（第一个字型）的编码为0x80、0x40、0x3e、0x10、0x20、0x20、0x1e、0x20；第二个字型编码为0x40、0x3e、0x10、0x20、0x20、0x1e、0x20、0x80，也就是把第一个字型编码中，第1行显示数据，变为第8行显示数据；第2行显示数据，变为第1行显示数据；第3行显示数据，变为第2行显示数据；第4行显示数据，变为第3行显示数据，依次类推。在此将利用阵列来储存这些数据，如下所示：

```
unsigned char d[8]={   0x80,   // 10000000
                       0x40,   // 01000000
                       0x3e,   // 00111110
                       0x10,   // 00010000
                       0x20,   // 00100000
                       0x20,   // 00100000
                       0x1e,   // 00011110
                       0x20 }; // 00100000
```

以图12-14的驱动电路为例，显示信号由P2送到LED点阵列的R1到R8，扫描信号由P1送到LED点阵列的C1到C8。如图12-15所示，整个图形的移动可看成显示8个字型，显示第一个字型，持续一段时间后，显示第二个字型，持续一段时间后，显示第三个字型……，如此看起来，就会感觉这个“μ”在往左边移动一样。而其中的“持续一段时间”将会决定字型跑的速度，这个时间越长，字型跑的速度越慢；这个时间越短，字型跑的速度越快。

在此以双循环的技巧，达到移动的目的，如下所示：

```
for(j=0;j<8;j++)                    // 8组数据
  {   scan=0x80;                    // 初始扫描信号
      for(i=0;i<8;i++)              // 扫描周期
      { if(i<j)P2=d[j-i-1];         // 若i<j则输出d[j-i-1]
```

```
            else  P2=d[7+(j-i)];                    // 若i>=j则输出d[7+(j-i)]
            P1=~scan;                               // 输出扫描信号
            delay1ms(2);                            // 延迟2ms
            scan>>=1;                               // 下一个扫描信号
            }                                       // 结束1个扫描周期
    }                                               // 结束
```

上述程序将从第1个字型（**j=0**）开始显示，如下所示：

当j=0，i=0时，P2=d[7+j-i]=d[7]=0x20，scan=0x80=10000000，P1=~scan=01111111。

当j=0，i=1时，P2=d[7+j-i]=d[6]=0x1e，scan=0x40=01000000，P1=~scan=10111111。

当j=0，i=2时，P2=d[7+j-i]=d[5]=0x20，scan=0x20=00100000，P1=~scan=11011111。

当j=0，i=3时，P2=d[7+j-i]=d[4]=0x20，scan=0x10=00010000，P1=~scan=11101111。

当j=0，i=4时，P2=d[7+j-i]=d[3]=0x10，scan=0x08=00001000，P1=~scan=11110111。

当j=0，i=5时，P2=d[7+j-i]=d[2]=0x3e，scan=0x04=00000100，P1=~scan=11111011。

当j=0, i=6时，P2=d[7+j-i]=d[1]=0x40，scan=0x02=00000010，P1=~scan=11111101。

当j=0，i=7时，P2=d[7+j-i]=d[0]=0x80，scan=0x01=00000001，P1=~scan=11111110。

紧接着是第2个字型（**j=1**），如下所示：

当j=1，i=0时，i<j，P2=d[j-i-1]=d[0]=0x80，P1=~scan=01111111。

当j=1，i=1时，P2=d[7+j-i]=d[7]=0x20，P1=~scan=10111111。

当j=1，i=2时，P2=d[7+j-i]=d[6]=0x1e，P1=~scan=11011111。

当j=1，i=3时，P2=d[7+j-i]=d[5]=0x20，P1=~scan=11101111。

当j=1，i=4时，P2=d[7+j-i]=d[4]=0x20，P1=~scan=11110111。

当j=1，i=5时，P2=d[7+j-i]=d[3]=0x10，P1=~scan=11111011。

当j=1，i=6时，P2=d[7+j-i]=d[2]=0x3e，P1=~scan=11111101。

当j=1，i=7时，P2=d[7+j-i]=d[1]=0x40，P1=~scan=11111110。

紧接着是第3个字型（**j=2**），如下所示：

当j=2，i=0时，i<j，P2=d[j-i-1]=d[1]=0x40，P1=~scan=01111111。

当j=2，i=1时，i<j，P2=d[j-i-1]=d[0]=0x80，P1=~scan=10111111。

当j=2，i=2时，P2=d[7+j-i]=d[7]=0x20，P1=~scan=11011111。

当j=2，i=3时，P2=d[7+j-i]=d[6]=0x1e，P1=~scan=11101111。

当j=2，i=4时，P2=d[7+j-i]=d[5]=0x20，P1=~scan=11110111。

当j=2，i=5时，P2=d[7+j-i]=d[4]=0x20，P1=~scan=11111011。

当j=2，i=6时，P2=d[7+j-i]=d[3]=0x10，P1=~scan=11111101。

当j=2，i=7时，P2=d[7+j-i]=d[2]=0x3e，P1=~scan=11111110。

紧接着是第4个字型（**j=3**），如下所示：

当j=3，i=0时，i<j，P2=d[j-i-1]=d[2]=0x3e，P1=~scan=01111111。

当j=3，i=1时，i<j，P2=d[j-i-1]=d[1]=0x40，P1=~scan=10111111。

当j=3，i=2时，i<j，P2=d[j-i-1]=d[0]=0x80，P1=~scan=11011111。

当j=3，i=3时，P2=d[7+j-i]=d[7]=0x20，P1=~scan=11101111。

当j=3，i=4时，P2=d[7+j-i]=d[6]=0x1e，P1=~scan=11110111。
当j=3，i=5时，P2=d[7+j-i]=d[5]=0x20，P1=~scan=11111011。
当j=3，i=6时，P2=d[7+j-i]=d[4]=0x20，P1=~scan=11111101。
当j=3，i=7时，P2=d[7+j-i]=d[3]=0x10，P1=~scan=11111110。
紧接着是第5个字型（**j=4**），如下所示：
当j=4，i=0时，i<j，P2=d[j-i-1]=d[3]=0x10，P1=~scan=01111111。
当j=4，i=1时，i<j，P2=d[j-i-1]=d[2]=0x3e，P1=~scan=10111111。
当j=4，i=2时，i<j，P2=d[j-i-1]=d[1]=0x40，P1=~scan=11011111。
当j=4，i=3时，i<j，P2=d[j-i-1]=d[0]=0x80，P1=~scan=11101111。
当j=4，i=4时，P2=d[7+j-i]=d[7]=0x20，P1=~scan=11110111。
当j=4，i=5时，P2=d[7+j-i]=d[6]=0x1e，P1=~scan=11111011。
当j=4，i=6时，P2=d[7+j-i]=d[5]=0x20，P1=~scan=11111101。
当j=4，i=7时，P2=d[7+j-i]=d[4]=0x20，P1=~scan=11111110。
紧接着是第6个字型（**j=5**），如下所示：
当j=5，i=0时，i<j，P2=d[j-i-1]=d[4]=0x20，P1=~scan=01111111。
当j=5，i=1时，i<j，P2=d[j-i-1]=d[3]=0x10，P1=~scan=10111111。
当j=5，i=2时，i<j，P2=d[j-i-1]=d[2]=0x3e，P1=~scan=11011111。
当j=5，i=3时，i<j，P2=d[j-i-1]=d[1]=0x40，P1=~scan=11101111。
当j=5，i=4时，i<j，P2=d[j-i-1]=d[0]=0x80，P1=~scan=11110111。
当j=5，i=5时，P2=d[7+j-i]=d[7]=0x20，P1=~scan=11111011。
当j=5，i=6时，P2=d[7+j-i]=d[6]=0x1e，P1=~scan=11111101。
当j=5，i=7时，P2=d[7+j-i]=d[5]=0x20，P1=~scan=11111110。
紧接着是第7个字型（**j=6**），如下所示：
当j=6，i=0时，i<j，P2=d[j-i-1]=d[5]=0x20，P1=~scan=01111111。
当j=6，i=1时，i<j，P2=d[j-i-1]=d[4]=0x20，P1=~scan=10111111。
当j=6，i=2时，i<j，P2=d[j-i-1]=d[3]=0x10，P1=~scan=11011111。
当j=6，i=3时，i<j，P2=d[j-i-1]=d[2]=0x3e，P1=~scan=11101111。
当j=6，i=4时，i<j，P2=d[j-i-1]=d[1]=0x40，P1=~scan=11110111。
当j=6，i=5时，i<j，P2=d[j-i-1]=d[0]=0x80，P1=~scan=11111011。
当j=6，i=6时，P2=d[7+j-i]=d[7]=0x20，P1=~scan=11111101。
当j=6，i=7时，P2=d[7+j-i]=d[6]=0x1e，P1=~scan=11111110。
紧接着是第8个字型（**j=7**），如下所示：
当j=7，i=0时，i<j，P2=d[j-i-1]=d[6]=0x1e，P1=~scan=01111111。
当j=7，i=1时，i<j，P2=d[j-i-1]=d[5]=0x20，P1=~scan=10111111。
当j=7，i=2时，i<j，P2=d[j-i-1]=d[4]=0x20，P1=~scan=11011111。
当j=7，i=3时，i<j，P2=d[j-i-1]=d[3]=0x10，P1=~scan=11101111。
当j=7，i=4时，i<j，P2=d[j-i-1]=d[2]=0x3e，P1=~scan=11110111。
当j=7，i=5时，i<j，P2=d[j-i-1]=d[1]=0x40，P1=~scan=11111011。
当j=7，i=6时，i<j，P2=d[j-i-1]=d[0]=0x80，P1=~scan=11111101。

当j=7，i=7时，P2=d[7+j-i]=d[7]=0x20，P1=~scan=11111110。

刚才的程序里，每个字型只扫描一周大概需要花16ms，这样字型会跑得很快，快到看不清楚！我们可多加一个k循环，让每个字型扫描10次，则约每0.16s移动一次，如下所示。

```
for(j=0;j<8;j++)                    // 8组数据
  for(k=0;k<10;k++)                 // 扫描10周
  {    scan=0x80;                   // 初始扫描信号
       for(i=0;i<8;i++)             // 扫描周期
       { if(i<j)P2=d[j-i-1];        // 若i<j则输出d[j-i-1]
         else  P2=d[7+(j-i)];       // 若i>=j则输出d[7+(j-i)]
         P1=~scan;                  // 输出扫描信号
         delay1ms(2);               // 延迟2ms
         scan>>=1;                  // 下个扫描信号
         }                          // 结束1个扫描周期
  }                                 // 结束10个扫描周期
```

若在程序的开头声明一个speed变数，以此变数作为移动速度的调整，其数值越大，移动速度越慢，如下所示。

```
unsigned char  speed=10;                  // 声明速度变数
:
:
for(j=0;j<8;j++)                          // 8组数据
 for(k=0;k<speed;k++)                     // 扫描speed周
 {    scan=0x80;                          // 初始扫描信号
      for (i=0;i<8;i++)                   // 扫描周期
      { if(i<j)P2=d[j-i-1];               // 若i<j则输出d[j-i-1]
        else  P2=d[7+(j-i)];              // 若i>=j则输出d[7+（j-i）]
        P1=~scan;                         // 输出扫描信号
        delay1ms(2);                      // 延迟2ms
        scan>>=1;                         // 下个扫描信号
        }                                 // 结束1个扫描周期
 }                                        // 结束10个扫描周期
```

右移

基本上，右移与左移的操作类似，只是扫描顺序相反，如图12-16所示，原本的字型（第一个字型），其编码为0x80、0x40、0x3e、0x10、0x20、0x20、0x1e、0x20；第二个字型编码为0x20、0x80、0x40、0x3e、0x10、0x20、0x20、0x1e，也就是把第一个字型编码中，第8行显示数据，变为第1行显示数据；第1行显示数据，变为第2行显示数据；

第2行显示数据，变为第3行显示数据；第3行显示数据，变为第4行显示数据，依次类推。

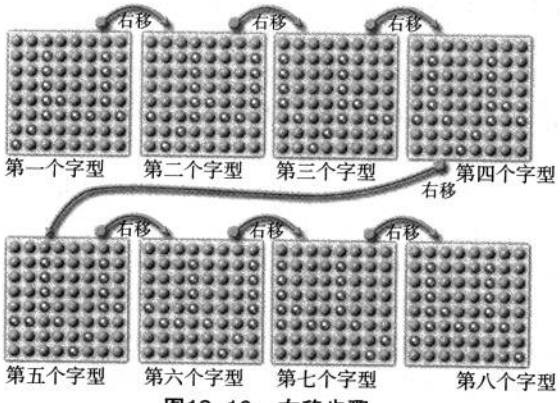


图12-16 右移步骤

同样以双循环的技巧，达到移动的目的，如下所示：

```
for(j=7;j<0;j--)                                    // 8组数据
  {    scan=0x01;                                   // 初始扫描信号
       for(i=7;i<0;i--)                             // 扫描周期
       { if(i>j)P2=d[8+（j-i）];                    // 若i>j则输出d[8+(j-i)]
         else  P2=d[j-i];                           // 若i<=j则输出d[j-i]
         P1=~scan;                                  // 输出扫描信号
         delay1ms(2);                               // 延迟2ms
         scan<<=1;                                  // 下个扫描信号
        }                                           // 结束1个扫描周期
  }                                                 // 结束
```

上述程序将从第1个字型（**j=7**）开始显示，如下所示。

当j=7，i=7时，P2=d[j-i]=d[0]=0x80，scan=0x01=00000001，P1=~scan=11111110。

当j=7，i=6时，P2=d[j-i]=d[1]=0x40，scan=0x02=00000010，P1=~scan=11111101。

当j=7，i=5时，P2=d[j-i]=d[2]=0x3e，scan=0x04=00000100，P1=~scan=11111011。

当j=7，i=4时，P2=d[j-i]=d[3]=0x10，scan=0x08=00001000，P1=~scan=11110111。

当j=7，i=3时，P2=d[j-i]=d[4]=0x20，scan=0x10=00010000，P1=~scan=11101111。

当j=7，i=2时，P2=d[j-i]=d[5]=0x20，scan=0x20=00100000，P1=~scan=11011111。

当j=7，i=1时，P2=d[j-i]=d[6]=0x1e，scan=0x40=01000000，P1=~scan=10111111。

当j=7，i=0时，P2=d[j-i]=d[7]=0x20，scan=0x80=10000000，P1=~scan=01111111。

紧接着是第2个字型（**j=6**），如下所示。

当j=6，i=7时，i>j，P2=d[8+（j-i）]=d[7]=0x20，P1=~scan=11111110。

当j=6，i=6时，P2=d[j-i]=d[0]=0x80，P1=~scan=11111101。

当j=6，i=5时，P2=d[j-i]=d[1]=0x40，P1=~scan=11111011。

当j=6，i=4时，P2=d[j-i]=d[2]=0x3e，P1=~scan=11110111。
当j=6，i=3时，P2=d[j-i]=d[3]=0x10，P1=~scan=11101111。
当j=6，i=2时，P2=d[j-i]=d[4]=0x20，P1=~scan=11011111。
当j=6，i=1时，P2=d[j-i]=d[5]=0x20，P1=~scan=10111111。
当j=6，i=0时，P2=d[j-i]=d[6]=0x1e，P1=~scan=01111111。
紧接着是第3个字型（**j=5**），如下所示。
当j=5，i=7时，i>j，P2=d[8+（j-i）]=d[6]=0x1e，P1=~scan=11111110。
当j=5，i=6时，i>j，P2=d[8+（j-i）]=d[7]=0x20，P1=~scan=11111101。
当j=5，i=5时，P2=d[j-i]=d[0]=0x80，P1=~scan=11111011。
当j=5，i=4时，P2=d[j-i]=d[1]=0x40，P1=~scan=11110111。
当j=5，i=3时，P2=d[j-i]=d[2]=0x3e，P1=~scan=11101111。
当j=5，i=2时，P2=d[j-i]=d[3]=0x10，P1=~scan=11011111。
当j=5，i=1时，P2=d[j-i]=d[4]=0x20，P1=~scan=10111111。
当j=5，i=0时，P2=d[j-i]=d[5]=0x20，P1=~scan=01111111。
紧接着是第4个字型（**j=4**），如下所示。
当j=4，i=7时，i>j，P2=d[8+（j-i）]=d[5]=0x20，P1=~scan=11111110。
当j=4，i=6时，i>j，P2=d[8+（j-i）]=d[6]=0x1e，P1=~scan=11111101。
当j=4，i=5时，i>j，P2=d[8+（j-i）]=d[7]=0x20，P1=~scan=11111011。
当j=4，i=4时，P2=d[j-i]=d[0]=0x80，P1=~scan=11110111。
当j=4，i=3时，P2=d[j-i]=d[1]=0x40，P1=~scan=11101111。
当j=4，i=2时，P2=d[j-i]=d[2]=0x3e，P1=~scan=11011111。
当j=4，i=1时，P2=d[j-i]=d[3]=0x10，P1=~scan=10111111。
当j=4，i=0时，P2=d[j-i]=d[4]=0x20，P1=~scan=01111111。
紧接着是第5个字型（**j=3**），如下所示。
当j=3，i=7时，i>j，P2=d[8+（j-i）]=d[4]=0x20，P1=~scan=11111110。
当j=3，i=6时，i>j，P2=d[8+（j-i）]=d[5]=0x20，P1=~scan=11111101。
当j=3，i=5时，i>j，P2=d[8+（j-i）]=d[6]=0x1e，P1=~scan=11111011。
当j=3，i=4时，i>j，P2=d[8+（j-i）]=d[7]=0x20，P1=~scan=11110111。
当j=3，i=3时，P2=d[j-i]=d[0]=0x80，P1=~scan=11101111。
当j=3，i=2时，P2=d[j-i]=d[1]=0x40，P1=~scan=11011111。
当j=3，i=1时，P2=d[j-i]=d[2]=0x3e，P1=~scan=10111111。
当j=3，i=0时，P2=d[j-i]=c[3]=0x10，P1=~scan=01111111。
紧接着是第6个字型（**j=2**），如下所示。
当j=2，i=7时，i>j，P2=d[8+（j-i）]=d[3]=0x10，P1=~scan=11111110。
当j=2，i=6时，i>j，P2=d[8+（j-i）]=d[4]=0x20，P1=~scan=11111101。
当j=2，i=5时，i>j，P2=d[8+（j-i）]=d[5]=0x20，P1=~scan=11111011。
当j=2，i=4时，i>j，P2=d[8+（j-i）]=d[6]=0x1e，P1=~scan=11110111。
当j=2，i=3时，i>j，P2=d[8+（j-i）]=d[7]=0x20，P1=~scan=11101111。
当j=2，i=2时，P2=d[j-i]=d[0]=0x80，P1=~scan=11011111。

当j=2，i=1时，P2=d[j-i]=d[1]=0x40，P1=~scan=10111111。
当j=2，i=0时，P2=d[j-i]=d[2]=0x3e，P1=~scan=01111111。
紧接着是第7个字型（**j=1**），如下所示。
当j=1，i=7时，i>j，P2=d[8+（j-i）]=d[2]=0x3e，P1=~scan=11111110。
当j=1，i=6时，i>j，P2=d[8+（j-i）]=d[3]=0x10，P1=~scan=11111101。
当j=1，i=5时，i>j，P2=d[8+（j-i）]=d[4]=0x20，P1=~scan=11111011。
当j=1，i=4时，i>j，P2=d[8+（j-i）]=d[5]=0x20，P1=~scan=11110111。
当j=1，i=3时，i>j，P2=d[8+（j-i）]=d[6]=0x1e，P1=~scan=11101111。
当j=1，i=2时，i>j，P2=d[8+（j-i）]=d[7]=0x20，P1=~scan=11011111。
当j=1，i=1时，P2=d[j-i]=d[0]=0x80，P1=~scan=10111111。
当j=1，i=0时，P2=d[j-i]=d[1]=0x40，P1=~scan=01111111。
紧接着是第8个字型（**j=0**），如下所示。
当j=0，i=7时，i>j，P2=d[8+（j-i）]=d[1]=0x40，P1=~scan=11111110。
当j=0，i=6时，i>j，P2=d[8+（j-i）]=d[2]=0x3e，P1=~scan=11111101。
当j=0，i=5时，i>j，P2=d[8+（j-i）]=d[3]=0x10，P1=~scan=11111011。
当j=0，i=4时，i>j，P2=d[8+（j-i）]=d[4]=0x20，P1=~scan=11110111。
当j=0，i=3时，i>j，P2=d[8+（j-i）]=d[5]=0x20，P1=~scan=11101111。
当j=0，i=2时，i>j，P2=d[8+（j-i）]=d[6]=0x1e，P1=~scan=11011111。
当j=0，i=1时，i>j，P2=d[8+（j-i）]=d[7]=0x20，P1=~scan=10111111。
当j=0，i=0时，P2=d[j-i]=d[0]=0x80，P1=~scan=01111111。

同样地，加入一个k循环及speed变数，让每个字型扫描speed次，调整此变数，即可调整移动的速度，其数值越大，移动速度越慢，如下所示。

```
unsigned char  speed=10;                    // 声明速度变数
:
:
for(j=7;j<0;j--)                            // 8组数据
  for(k=0;k<speed;k++)                      // 扫描speed周
  {     scan=0x01;                          // 初始扫描信号
        for(i=7;i<0;i--)                    // 扫描周期
        { if(i>j)P2=d[8+(j-i)];             // 若i>j则输出d[8+(j-i)]
          else(i<=j)P2=d[j-i];              // 若i<=j则输出d[j-i]
          P1=~scan;                         // 输出扫描信号
          delay1ms(2);                      // 延迟2ms
          scan<<=1;                         // 下个扫描信号
         }                                  // 结束1个扫描周期
};                                          // 结束10个扫描周期
```

12.4.2 垂直移动

若要将文字或图形在LED阵列上下移动，只要改变每行显示数据的顺序即可，以如图12-17所示的编码为例：

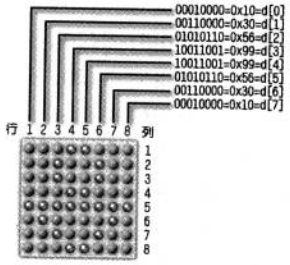


图12-17 图形的编码

若按第1行由原本的“00010000”改变为“00100000”；第2行由原本的“00110000”改变为“01100000”；第3行由原本的“01010110”改变为“10101100”，依次类推，则显示这些新数据时，就有图样下卷的感觉。简言之，只要利用每组数据的左移或右移，即可产生“下卷”、“上卷”的感觉。

上卷

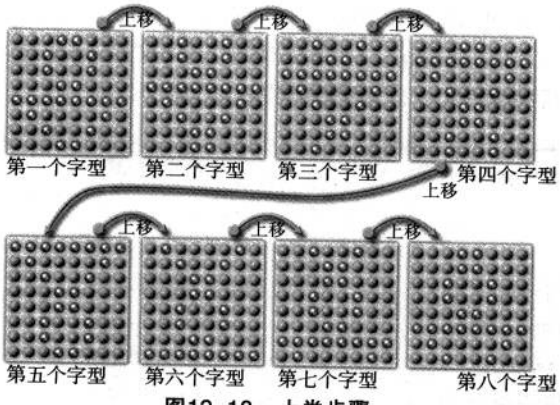


图12-18 上卷步骤

如图12-18所示，8×8LED阵列的上卷就是显示8个不同的字型。首先扫描第一个字型，同样是8行、8次扫描、8次显示；完成第一个字型后，再扫描第二个字型；完成第二个字型后，再扫描第三个字型，依次类推；即可产生该文字或图形上卷的感觉。

原本的字型（第一个字型），其编码为00010000、00110000、01010110、10011001、10011001、01010110、00110000、000100000；第二个字型编码为00001000、00011000、00101011、11001100、11001100、00101011、00011000、000010000，也就是把第一个字型编码中，每行显示数据都右移一位，即可产生上卷的效果。在此利用阵列储存这些数据，如下：

```
unsigned char d[8]={    0x10,   // 00010000
                        0x30,   // 00110000
                        0x56,   // 01010110
                        0x99,   // 10011001
                        0x99,   // 10011001
                        0x56,   // 01010110
                        0x30,   // 00110000
                        0x10 }; // 00010000
```

此外，为保险起见，在此也声明一个disp[8]阵列，在开始显示之前，就把d[8]阵列的数据放入disp[8]里，即：

```
    for(i=0;i<8;i++)                    // 执行八次
        disp[i]=d[i];                   // 将d[]数据复制到disp[]
```

在组合语言里，“RR A”指令可将ACC暂存器的数据右移一位，而其最右边位将移至最左边。在C语言里，可没这么方便！若执行“disp[i]>>=1”指令时，可将disp[i]数据右移一位，但其最右边位并不会移至最左边，而是填入0。所以，在进行移位之前，必须先判断最右边位是什么？若判断最右边位为0，那就不需额外的处理，直接移位就好；若判断最右边位为1，在移位之后，还要再执行加0x80的处理，如下：

```
for(i=0;i<8;i++)                        // 8组数据
  if(disp[i]%2==1)                      // 最右边位为1
      {disp[i]>>=1;                     // 右移一位
       disp[i]+=0x80; }                 // 加上0x80
  else disp[i]>>=1;                     // 最右边位为0，直接右移一位
}                                       // 结束
```

数据处理完成后，即可输出。而输出时，可重复扫描如下：

```
for(k=0;k<speed;k++)                    // 重复执行k次
  for(i=0;i<8;i++)                      // 扫描一周
  {   P2=disp[i];                       // 输出显示信号
      P1=scan;                          // 输出扫描信号
      delay1ms(2);                      // 延迟2ms
      scan<<=1;                         // 下一个扫描信号
  }                                     // 扫描一周结束
}                                       // 结束
```

其中speed就是重复的次数，重复次数越多，上卷速度越慢；重复次数越少，上卷速度越快。

下卷

如图12-19所示，下卷与上卷类似，同样是显示8个不同的字型。首先扫描第一个字型，同样是8行、8次扫描、8次显示；完成第一个字型后，再扫描第二个字型；完成第二个字型后，再扫描第三个字型，依次类推，即可产生该文字或图形下卷的感觉。

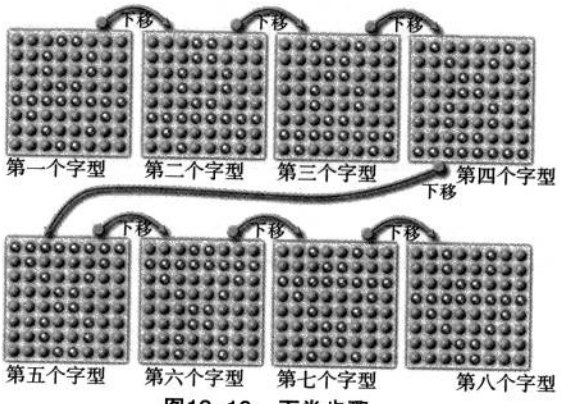


图12-19 下卷步骤

原本的字型（第一个字型），其编码为00010000、00110000、01010110、10011001、10011001、01010110、00110000、000100000；第二个字型编码为00100000、01100000、10101100、00110011、00110011、10101100、01100000、001000000，也就是把第一个字型编码中，每行显示数据都左移一位，即可产生下卷的效果。

同样地，在此也是以disp[8]阵列来操作，而在开始显示之前，就把d[8]阵列的数据放入disp[8]里，在此不赘述。在C语言里，同样必须面对执行“disp[i]<<=1”指令后，虽然disp[i]数据左移一位，但其最左边位并不会移至最右边，而是填入0。所以，在进行移位之前，必须先判断最左边位是什么？若判断最左边位为0，那就不须额外的处理，直接移位就好；若判断最左边位为1，在移位之后，还要再执行加0x01的处理，如下：

```
for(i=0;i<8;i++)                    // 8组数据
  if(disp[i]>=0x80)                 // 最左边位为1
       { disp[i]<<=1;               // 左移一位
         disp[i]+=0x01; }           // 加上0x80
  else disp[i]<<=1;                 // 最左边位为0，直接左移一位
}                                   // 结束
```

数据处理完成后，即可输出。而输出方式与上卷完全一样，如此一来，我们可将输出部分变成一个函数，程序会比较简单，如下：

```
display(unsigned char speed)                    // 显示函数开始
{ for(k=0;k<speed;k++)                          // 重复执行k次
      for(i=0;i<8;i++)                          // 扫描一周
      { P2=disp[i];                             // 输出显示信号
      P1=scan;                                  // 输出扫描信号
      delay1ms(2);                              // 延迟2ms
      scan<<=1; }                               // 下一个扫描信号
  }                                             // 扫描重复执行结束
}                                               // 结束
```

12.5 实例演练

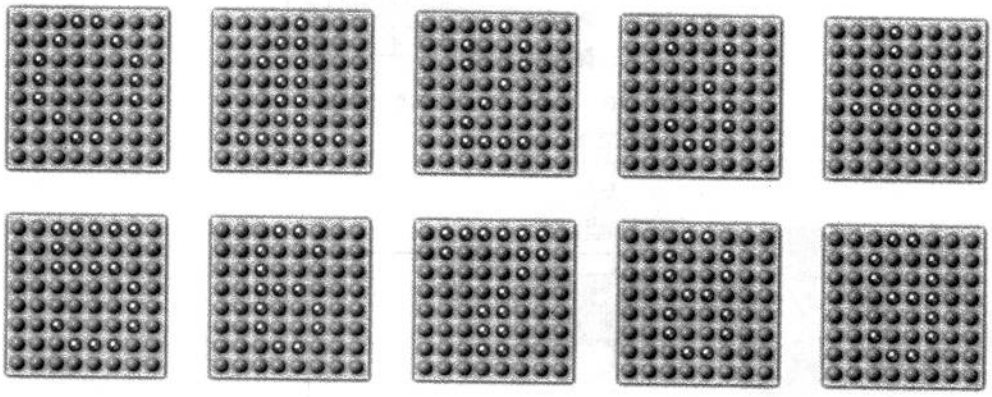


在本单元里提供多个LED阵列的应用范例，如下所示：

12.5.1 8 × 8LED阵列静态展示实例演练

● 实验要点

如图12-20所示，其中的8 × 8共阳极LED阵列可采用1.2英寸的MM12884P或2.3英寸的MM23884P。扫描信号连接到8051的Port 1，显示信号连接到8051的Port 2。在此将由这个LED阵列显示0~9，约每0.5s增加1。0~9的编码如下：



位数	1	2	3	4	5	6	7	8
0	0x00	0x1c	0x22	0x41	0x41	0x22	0x1c	0x00
1	0x00	0x40	0x44	0x7e	0x7f	0x40	0x40	0x00
2	0x00	0x00	0x66	0x51	0x49	0x46	0x00	0x00
3	0x00	0x00	0x22	0x41	0x49	0x36	0x00	0x00

续表

位数	1	2	3	4	5	6	7	8
4	0x00	0x10	0x1c	0x13	0x7c	0x7c	0x10	0x00
5	0x00	0x00	0x27	0x45	0x45	0x45	0x39	0x00
6	0x00	0x00	0x3e	0x49	0x49	0x32	0x00	0x00
7	0x00	0x03	0x01	0x71	0x79	0x07	0x03	0x00
8	0x00	0x00	0x36	0x49	0x49	0x36	0x00	0x00
9	0x00	0x00	0x26	0x49	0x49	0x3e	0x00	0x00

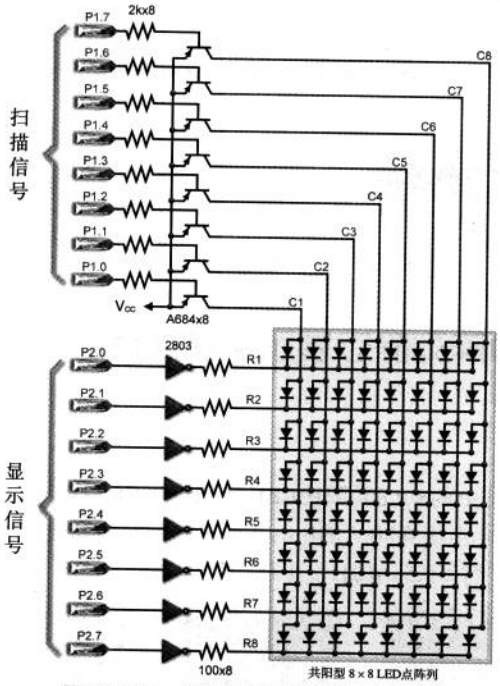
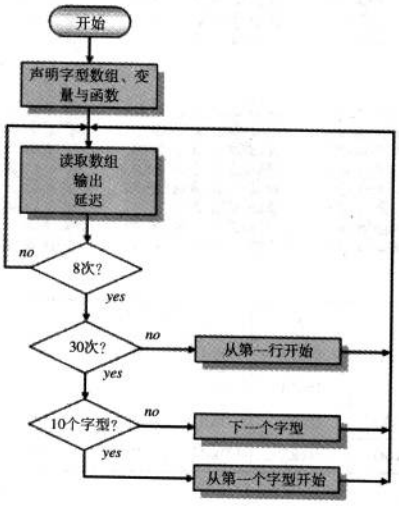


图12-20 单色8×8 LED实验电路图



流程图与程序设计

```
/* 8×8 LED阵列实验(ch12-1.c) */
#include <reg51.h>
#define ROWP  P2                          // 输出列接至P2
#define COLP  P1                          // 扫描行接至P1
unsigned char disp[][8]=          // ======= 字 型 ==============
{ {0x00, 0x1c, 0x22, 0x41, 0x41, 0x22, 0x1c, 0x00},      // 0
  {0x00, 0x40, 0x44, 0x7e, 0x7f, 0x40, 0x40, 0x00},      // 1
  {0x00, 0x00, 0x66, 0x51, 0x49, 0x46, 0x00, 0x00},      // 2
  {0x00, 0x00, 0x22, 0x41, 0x49, 0x36, 0x00, 0x00},      // 3
  {0x00, 0x10, 0x1c, 0x13, 0x7c, 0x7c, 0x10, 0x00},      // 4
  {0x00, 0x00, 0x27, 0x45, 0x45, 0x45, 0x39, 0x00},      // 5
  {0x00, 0x00, 0x3e, 0x49, 0x49, 0x32, 0x00, 0x00},      // 6
  {0x00, 0x03, 0x01, 0x71, 0x79, 0x07, 0x03, 0x00},      // 7
  {0x00, 0x00, 0x36, 0x49, 0x49, 0x36, 0x00, 0x00},      // 8
  {0x00, 0x00, 0x26, 0x49, 0x49, 0x3e, 0x00, 0x00}};     // 9
unsigned char speed=30;                   // 约0.48s
void delay1ms(unsigned char);             // 声明延迟函数
//============== 主 程 序 ==================
main()                                    // 主程序开始
{   unsigned char i,j,k,scan;             // 声明变数
    while(1)                              // 无穷尽循环
        for(i=0;i<10;i++)                 // 字型循环
          for(k=0;k<speed;k++)            // 重复执行k次
          {       scan=1;                 // 初始扫描信号
                  for(j=0;j<8;j++)        // 扫描一周
                  {   ROWP=0XFF;          // 关闭LED
                      COLP=~scan;         // 输出扫描信号
                      ROWP=~disp[i][j];       // 输出显示信号
                      delay1ms(2);        // 延迟2ms
                      scan<<=1;           // 下一个扫描信号
                  }                       // j循环结束
          }                               // k循环结束
}                                         // 主程序结束
//============== 延迟函数 ==================
void delay1ms(unsigned char x)
{   int i,j;                              // 声明变数
```

```
    for(i=0;i<x;i++)                    // 外循环
        for(j=0;j<120;j++);             // 内循环
}                                       // 延迟函数结束
```

8×8 LED阵列静态演示实验（ch12-1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图12-20连接线路，使用在线仿真器，载入新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

1. 请试着更改本实验里的电路，改采8×8用共阴型LED阵列，再执行同样的功能？

2. 请试着更改本实验里的程序，让LED阵列依次显示你的出生年月日？

12.5.2 8×8LED阵列多色显示

实验要点

如图12-26所示，其中MM12883AP为绿/橙双色1.2英寸8×8共阴型LED阵列，其中包括绿色与橙色两组扫描信号，分别连接到8051的Port 1与Port 3，显示信号则连接到

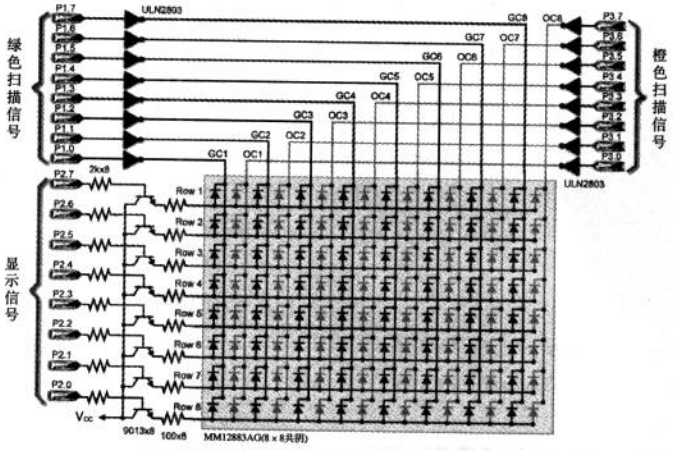
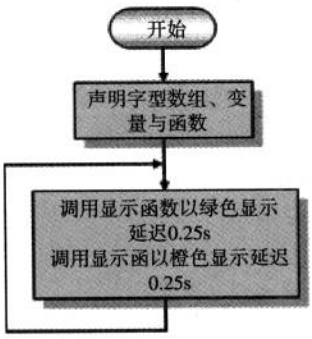
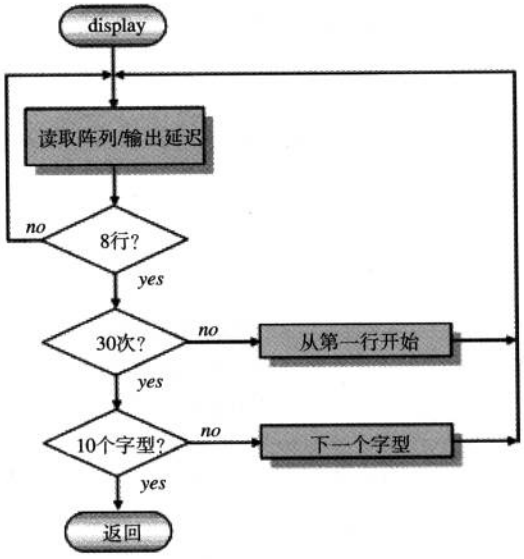


图12-21 多色显示实验电路图

8051的Port 2。在本单元里，将由这个LED阵列显示0~9，约每0.5s增加1，第一轮以绿色显示，第二轮以橙色显示；而0~9的编码与12.5.1节一样。





流程图与程序设计

```
/* 8 x 8 LED彩色阵列实验(ch12-2.c) */
#include <reg51.h>
#define ROWP  P2                          // 输出列接至P2
#define green  P1                         // 定义绿色行扫描信号连接口
#define orange  P3                        // 定义橙色行扫描信号连接口
unsigned char disp[][8]=          // ====== 字 型 ==========
{ {0x00, 0x1c, 0x22, 0x41, 0x41, 0x22, 0x1c, 0x00},       // 0
  {0x00, 0x40, 0x44, 0x7e, 0x7f, 0x40, 0x40, 0x00},       // 1
  {0x00, 0x00, 0x66, 0x51, 0x49, 0x46, 0x00, 0x00},       // 2
  {0x00, 0x00, 0x22, 0x41, 0x49, 0x36, 0x00, 0x00},       // 3
  {0x00, 0x10, 0x1c, 0x13, 0x7c, 0x7c, 0x10, 0x00},       // 4
  {0x00, 0x00, 0x27, 0x45, 0x45, 0x45, 0x39, 0x00},       // 5
  {0x00, 0x00, 0x3e, 0x49, 0x49, 0x32, 0x00, 0x00},       // 6
  {0x00, 0x03, 0x01, 0x71, 0x79, 0x07, 0x03, 0x00},       // 7
  {0x00, 0x00, 0x36, 0x49, 0x49, 0x36, 0x00, 0x00},       // 8
  {0x00, 0x00, 0x26, 0x49, 0x49, 0x3e, 0x00, 0x00}};      // 9
//        unsigned char speed=30;           // 约0.48s
 void delay1ms(unsigned char);              // 声明延迟函数
 void display(unsigned char, bit);          // 声明显示函数
 //============== 主 程 序 ==================
```

```
main()                                          // 主程序开始
{   while(1)                                    // 无穷尽循环
    {    display(30,0);                         // 显示绿色
         ROWP=0x00;                             // 关闭LED
         delay1ms(250);                         // 延迟0.25s
         display(30,1);                         // 显示橙色
         ROWP=0x00;                             // 关闭LED
         delay1ms(250);                         // 延迟0.25s
    }                                           // while结束
}                                                       // 主程序结束/
//============== 显示函数 ==================
void display(unsigned char speed,bit color)         // 显示函数开始
{   unsigned char i,j,k,scan;                   // 声明变数
    for(i=0;i<10;i++)                           // 字型循环
       for(k=0;k<speed;k++)                     // 重复执行k次
       {     scan=0x01;                         // 初始扫描信号
             for(j=0;j<8;j++)                   // 扫描一周
             {   ROWP=0;                        // 关闭LED
                 if(color==0)                   // 若color=0
                       green=scan;              // 输出绿色扫描信号
                 else   orange= scan;           // color=1,输出橘色扫描信号
                 ROWP=~disp[i][j];              // 输出显示信号
                 delay1ms(2);                   // 延迟2ms
                 scan<<=1;                      // 下一个扫描信号
             }                                  // j循环结束
       }                                        // k循环结束
}                                               // 显示函数结束
//============== 延迟函数 ==================
void delay1ms(unsigned char x)
{ int    i,j;                                   // 声明变数
  for(i=0;i<x;i++)                              // 外循环
      for(j=0;j<120;j++);                       // 内循环
}                                               // 延迟函数结束
```

8×8 LED阵列多色显示实验（ch12-2.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。
2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状

况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图12-21连接线路，使用在线仿真器，载入新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

1. 在本实验里，若要多一组显示颜色（绿色+橙色），应如何修改？

2. 请试着更改本实验里的程序，让LED阵列依次显示你的出生年月日，其中以橙色显示年，绿色显示月，绿色+橙色显示日？

12.5.3 8×8LED阵列平移实例演练

实验要点

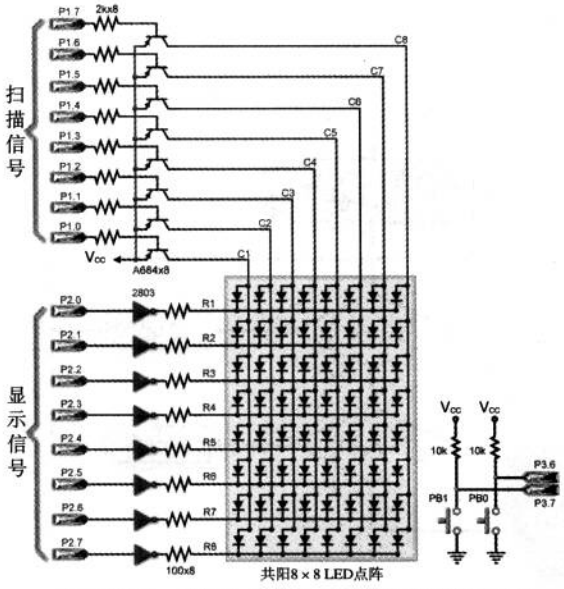


图12-22 单色8×8 LED动态移动实验电路图

如图12-22所示，其扫描信号连接到8051的Port 1，显示信号连接到8051的Port 2。按PB0钮后，向右箭头字型将右移三圈；按PB1钮后，向左箭头字型将左移三圈。而向右箭头字型与向左箭头字型，如图12-23所示。

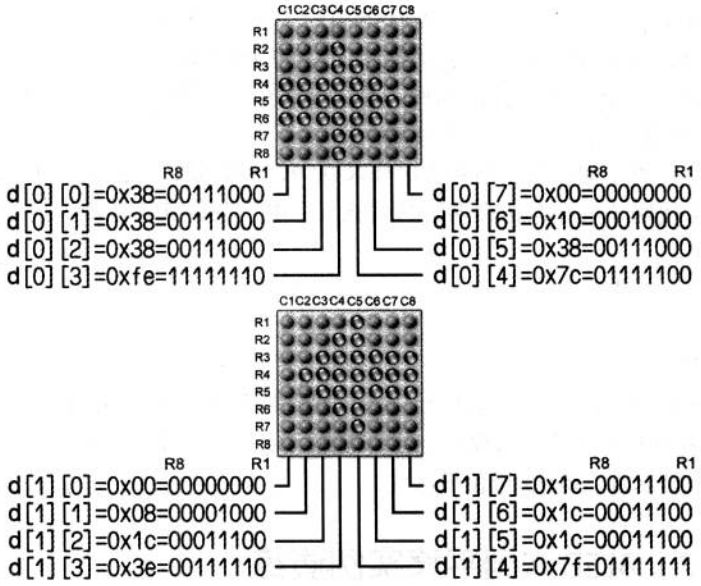
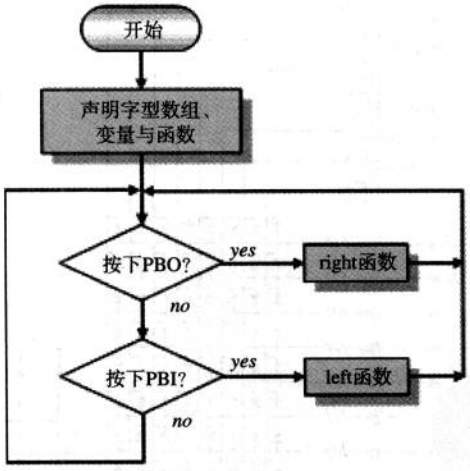
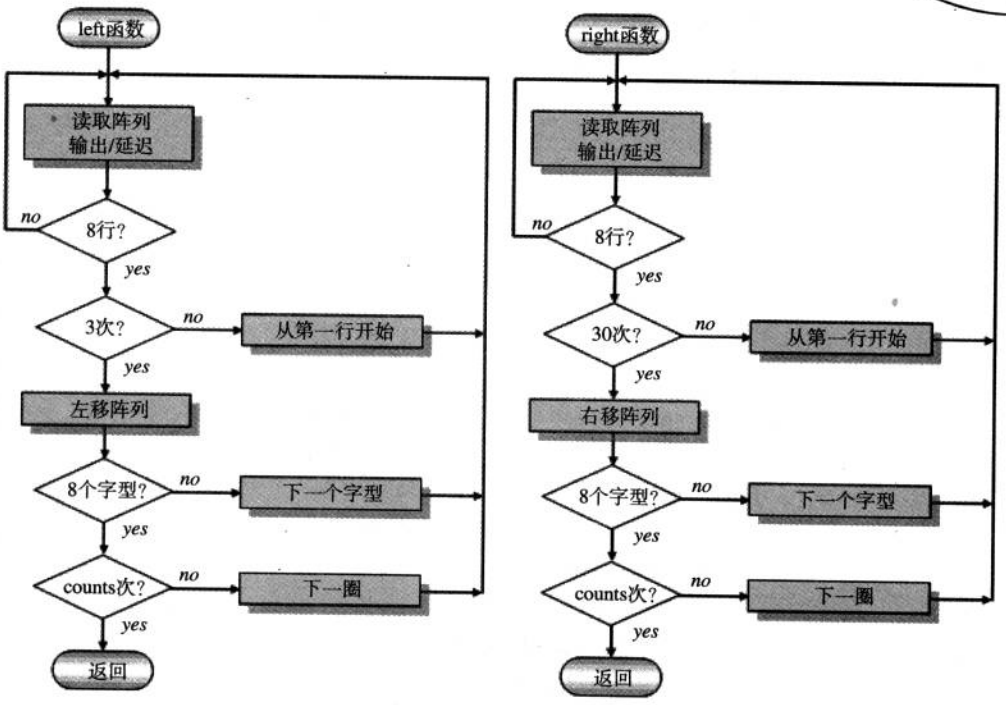


图12-23 向右箭头与向左箭头编码

流程图与程序设计



主程序




```
/* 8×8 LED阵列平移实验(ch12-3.c)*/
#include  <reg51.h>
#define ROWP   P2                          // 输出列接至P2
#define COLP   P1                          // 扫描行接至P1
//============== 字 型 ==================
unsigned char d[][]={     {0x38, 0x38, 0x38, 0xfe, 0x7c, 0x38, 0x10, 0x00},   // ➡
                          {0x00, 0x08, 0x1c, 0x3e, 0x7f, 0x1c, 0x1c, 0x1c}};  // ⬅
unsigned char speed=30;                    // 约0.48s
unsigned char scan;                        // 声明变数
sbit PB0 = P3^6;                           // 声明左移按钮
sbit PB1 = P3^7;                           // 声明右移按钮
void left(unsigned char, unsigned char);   // 声明左移函数
void right(unsigned char, unsigned char);// 声明右移函数
void delay1ms(unsigned char);    // 声明延迟函数
//============== 主 程 序 ==================
main()                                     // 主程序开始
{ P3=0xff;                                 // 设定Port 3为输入口
  while(1)                                 // 无穷尽循环
  {   if(PB0==0)right(0,3);                // 执行第一个字型，右移三圈
```

```
        if(PB1==0)left(1,3);// 执行第二个字型，左移三圈
        delay1ms(10);                          // 延迟10ms
        ROWP=0xff;                             // 关闭LED
    }                                          // while结束
}                                              // 主程序结束
//============== 右移函数 ==================
void right(unsigned char word,unsigned char counts)      // 右移函数开始
{ char  i,j,k,l;                               // 声明变数
  for(l=0;l<counts;l++)                        // 执行counts圈
    for(j=7;j>=0;j--)                          // 8组数据
       for(k=0;k<speed;k++)                    // 扫描speed周
       {   scan=0x01;                          // 初始扫描信号
           for(i=7;i>=0;i--)                   // 扫描周期
           {    ROWP=0xff;                     // 关闭LED
                COLP=~scan;                    // 输出扫描信号
                if(i>j)  ROWP=~d[word][8+(j-i)];
                /* 若i>j则输出~d[word][8+(j-i)] */
                else if(i<=j)ROWP=~d[word][j-i];
                /* 若i<=j则输出~d[word][j-i] */
                delay1ms(2);                   // 延迟2ms
                scan<<=1;                      // 下一个扫描信号
           }                                   // 结束1个扫描周期
       }                                       // 结束speed个扫描周期
}                                              // 结束右移函数
//============== 左移函数 ==================
void left(unsigned char word,unsigned char counts)// 左移函数开始
{ unsigned char  i,j,k,l;                      // 声明变数
  for(l=0;l<counts;l++)                        // 执行counts圈
    for(j=0;j<8;j++)                           // 8组数据
       for(k=0;k<speed;k++)                    // 扫描speed周
       {   scan=0x80;                          // 初始扫描信号
           for(i=0;i<8;i++)                    // 扫描周期
           {    ROWP=0xff;                     // 关闭LED
                COLP=~scan;                    // 输出扫描信号
                if(i<j)ROWP=~d[word][j-i-1];
                /* 若i<j则输出~d[word][j-i-1] */
                else  ROWP=~d[word] [7+(j-i)];
                /* 若i>=j则输出~d[word][7+(j-i)] */
```

```
                delay1ms(2);              // 延迟2ms
                scan>>=1;                 // 下个扫描信号
            }                             // 结束1个扫描周期
        }                                 // 结束speed个扫描周期
}                                         // 结束左移函数
//============== 延迟函数 ==================
void delay1ms(unsigned char x)
{ unsigned char i,j;                      // 声明变数
  for(i=0;i<x;i++)                        // 外循环
     for(j=0;j<120;j++);                  // 内循环
}                                         // 延迟函数结束
```

8×8 LED阵列平移实验（ch12-3.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图12-22连接线路，使用在线仿真器，载入新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

1. 在本实验里，若按住PB0不放，会怎样？

2. 修改本实验的程序，使得放开按钮后才进行左/右移的动作？

12.5.4 8×8LED阵列跑马灯实例演练

实验要点

如图12-20所示（12.5.1节），其扫描信号连接到8051的Port 1，显示信号连接到8051的Port 2。在本单元里将由这个LED阵列动态显示“8051”，即跑马灯文字幕，约每0.32s左移一步，如图12-24所示。

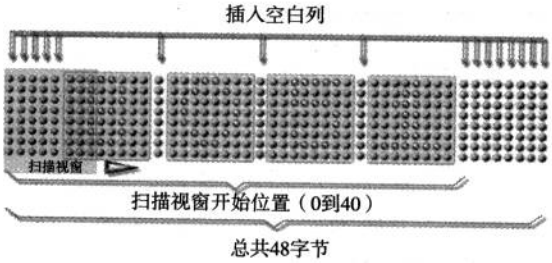
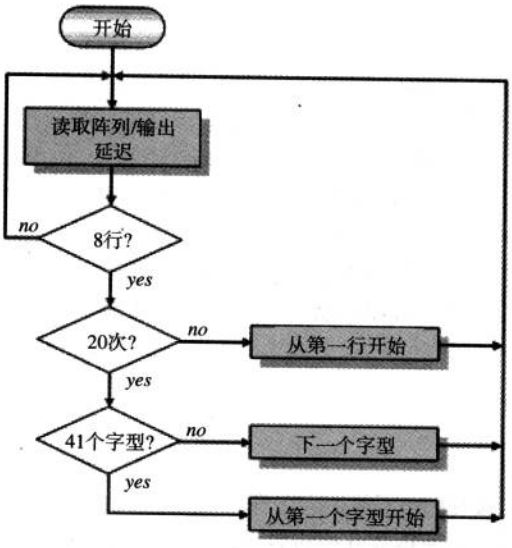


图12-24　显示功能示意图

如图12-24所示，每次扫描一个视窗（8条扫描线），从d[0]开始为第一个扫描视窗，第一个扫描视窗显示完毕后，再以d[1]开始为第二个扫描视窗，直到d[40]开始的扫描视窗显示完毕后，再由第一个扫描视窗开始扫描。因此，我们在d[]阵列里，总共放置48组显示数据。

流程图与程序设计




```
/* 8×8 LED阵列跑马灯实验-8051数字左移(ch12-4.c) */
#include <reg51.h>
#define ROWP        P2      // 输出列接至P2
#define COLP        P1      // 扫描行接至P1
//=============== 字 型 ==================
unsigned char d[]={  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x36,
                     0x49, 0x49, 0x36, 0x00, 0x00, 0x00, 0x00, 0x1c,
                     0x22, 0x41, 0x41, 0x22, 0x1c, 0x00, 0x00, 0x00,
```

```
                    0x00, 0x27, 0x45, 0x45, 0x39, 0x00, 0x00, 0x00,
                    0x00, 0x40, 0x44, 0x7e, 0x7f, 0x40, 0x40, 0x00,
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
unsigned char speed=20;                    // 约0.32s
void delay1ms(unsigned char);              // 声明延迟函数
//============== 主 程 序 ==================
main()                                     // 主程序开始
{ unsigned char i,j,k,scan;                // 声明变数
  while(1)                                 // 无穷尽循环
  {   for(i=0;i<41;i++)                    // 显示40个视窗
      {  for(k=0;k<speed;k++)              // 每个视窗重复执行k次
         {  scan=1;                        // 初始扫描信号
            for(j=0;j<8;j++)               // 扫描一周
            {  ROWP=0XFF;                  // LED OFF
               COLP=~scan;                 // 输出扫描信号
               ROWP=~d[i+j];               // 输出显示信号
               delay1ms(2);                // 延迟2ms
               scan<<=1;}                  // 下一个扫描信号
            }                              // 完成扫描一周
      }                                    // 完成k次
  }                                        // while叙述结束
}                                          // 主程序结束
//============== 延迟函数 ==================
void delay1ms(unsigned char x)
{ unsigned char i,j;                       // 声明变数
  for(i=0;i<x;i++)                         // 外循环
     for(j=0;j<120;j++);                   // 内循环
}                                          // 延迟函数结束
```

8×8 LED阵列跑马灯实验（ch12-4.c）

● 操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图12-20连接线路，使用在线仿真器，载入新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，

看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

1. 在本实验里，文字呈现动态左移，请将它修改为动态右移？
2. 根据本实验的电路，请编写一个电话号码的动态左移？

12.5.5 8×8LED阵列垂直移动实例演练

实验要点

如图12-22所示（12.5.3节），其扫描信号连接到8051的Port 1，显示信号连接到8051的Port 2。按PB0钮后，向上箭头字型将上移三圈；按PB1钮后，向下箭头字型将下移三圈。而向上箭头字型与向下箭头字型，如图12-25所示。

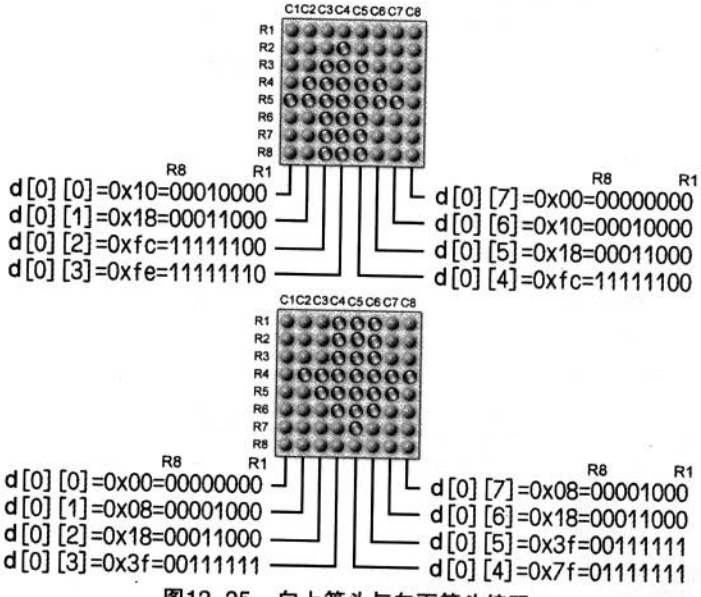
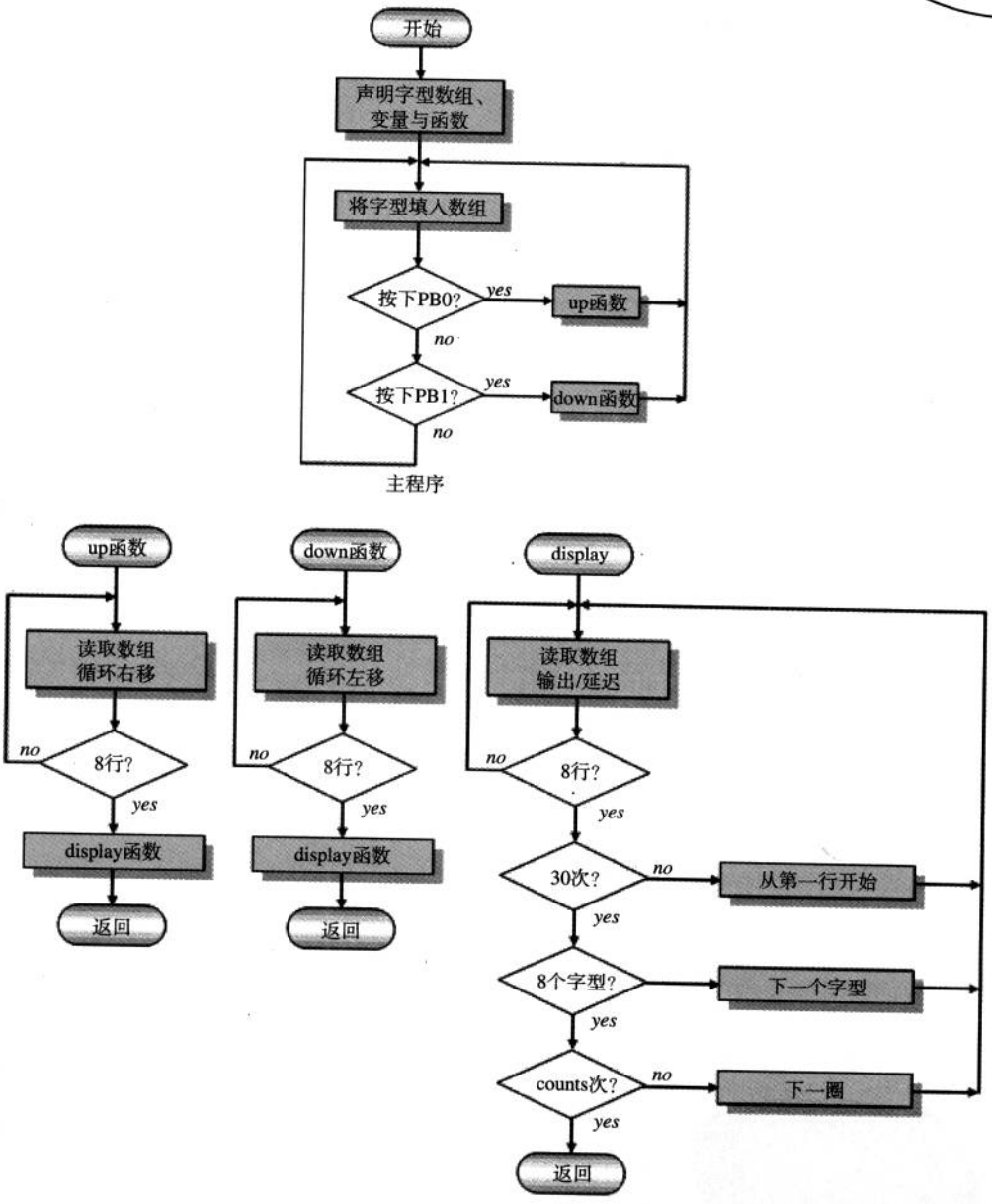


图12-25 向上箭头与向下箭头编码

流程图与程序设计




```
/* 8×8 LED阵列垂直移动实验(ch12-5.c)*/
#include <reg51.h>
#define ROWP  P2      // 输出列接至P2
#define COLP  P1      // 扫描行接至P1
//============== 字 型 ==================
```

```
unsigned char d[][]={    {0x10, 0x18, 0xfc, 0xfe, 0xfc, 0x18, 0x10, 0x00}, //
                         {0x00, 0x08, 0x18, 0x3f, 0x7f, 0x3f, 0x18, 0x08}};//
unsigned char disp[2][8];// 声明阵列
unsigned char speed=30;                          // 约0.48s
sbit PB0 = P3^6;                                 // 声明左移按钮
sbit PB1 = P3^7;                                 // 声明右移按钮
void up(unsigned char, unsigned char);           // 声明上移函数
void down(unsigned char, unsigned char);         // 声明下移函数
void display(unsigned char, unsigned char);      // 声明显示函数
void delay1ms(unsigned char);                    // 声明延迟函数
//============== 主 程 序 ==================
main()                                           // 主程序开始
{ unsigned char i,j;                             // 声明变数
  P3=0xff;                                       // 设定Port 3为输入口
  while(1)                                       // 无穷尽循环
  {   for(i=0;i<2;i++)                           // 外循环
        for(j=0;j<8;j++)                         // 内循环
            disp[i][j]=d[i][j];                  // 填入disp阵列
        if(PB0==0)up(0,3);                       // 执行第一个字型，上移三圈
        if(PB1==0)down(1,3);                     // 执行第二个字型，下移三圈
  }                                              // while结束
}                                                // 主程序结束
//============== 上移函数 ==================
void up(unsigned char word,unsigned char counts)// 上移函数开始
{ unsigned char  i,j,K;                          // 声明变数
  for(K=0;K<counts;K++)
  for(j=0;j<8;j++)
     for(i=0;i<8;i++)                            // 8组数据
     {    if(disp[word][i]%2==1)                 // 最右边位为1
           {   disp[word][i]>>=1;                // 右移一位
               disp[word][i]+=0x80; }            // 加上0x80
          else disp[word][i]>>=1;                // 最右边位为0，直接右移一位
          display(word,1);                       // 显示
  } }                                            // 结束
//============== 下移函数 ==================
void down(unsigned char word,unsigned char counts)//上移函数开始
{ unsigned char  i,j,K;                          // 声明变数
  for(K=0;K<counts;K++)
```

```
      for(j=0;j<8;j++)
        for(i=0;i<8;i++)                          // 8组数据
        {  if(disp[word][i]>=0x80)                // 最左边位为1
           { disp[word][i]<<=1;                   // 左移一位
             disp[word][i]+=0x01;                 // 加上0x01
           }
             else disp[word][i]<<=1;              // 最左边位为0，直接左移一位
             display(word,1);                     // 显示
 }         }                                      // 结束
 //============== 显示函数 ==================
 void display(unsigned char word,unsigned char counts)//显示函数开始
 { unsigned char  i,j,k,l,scan;                   // 声明变数
   for(l=0;l<counts;l++)                          // 执行counts圈
     for(j=0;j<8;j++)                             // 8组数据
       for(k=0;k<speed;k++)                       // 扫描speed周
       {   scan=0x80;                             // 初始扫描信号
           for(i=0;i<8;i++)                       // 扫描周期
           {   ROWP=0XFF;                         // LED OFF
               COLP=~scan;                        // 输出扫描信号
               ROWP=~disp[word][i];               // 输出显示信号
               delay1ms(2);                       // 延迟2ms
               scan>>=1;                          // 下个扫描信号
           }                                      // 结束1个扫描周期
       }                                          // 结束speed个扫描周期
 }                                                // 结束左移函数
 //============== 延迟函数 ==================
 void delay1ms(unsigned char x)
 { int  i,j;                                      // 声明变数
   for(i=0;i<x;i++)                               // 外循环
     for(j=0;j<120;j++);                          // 内循环
 }                                                // 延迟函数结束
```

8×8 LED阵列垂直移动实验（ch12-5.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图12-22连接线路，使用在线仿真器，载入新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

● 思考一下

在本实验里，若改用共阴极LED阵列，电路图应如何修改？

12.5.6 8 × 8LED阵列卷动实例演练

● 实验要点

如图12-20所示（12.5.1节），其中扫描信号连接到8051的Port 1，显示信号连接到8051的Port 2。在本单元里将由这个LED阵列动态显示“8051”，而其字型移动方向是由下而上移动（扫描视窗则是由上而下移动），每0.32s上卷一步。如图12-26所示，每次扫描一个视窗（8条扫描线），从d[0]开始为第一个扫描视窗，第一个扫描视窗显示完毕后，再以d[1]开始为第二个扫描视窗，直到d[40]开始的扫描视窗显示完毕后，再由第一个扫描视窗开始扫描。因此，我们在d[]阵列里，总共放置48组显示数据。

基本上，垂直卷动与跑马灯类似，只是垂直卷动的字型与跑马灯的字型，方向相差90°。我们可利用12.5.4节的跑马灯程序，将其字型顺时针旋转90°，而观看LED阵列时，则逆时针旋转90°，即可看出字型往上卷动，如图12-27所示。

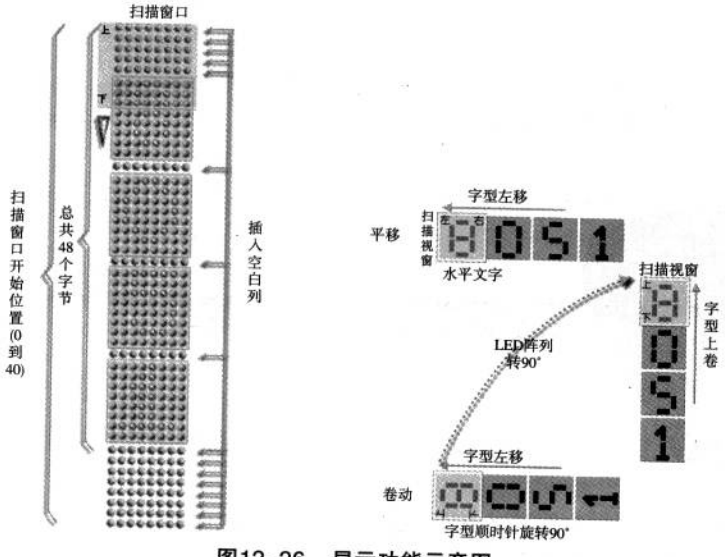
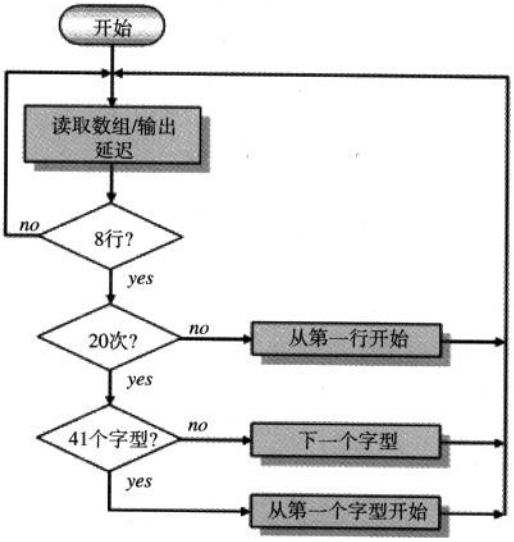


图12-26 显示功能示意图

流程图与程序设计




```
/* 8×8 LED阵列上卷实验(ch12-6.c) */
#include <reg51.h>
#define ROWP  P2                    // 输出列接至P2
#define COLP  P1                    // 扫描行接至P1
//============== 字 型 ==================
unsigned char d[][]={ 0x00, 0x00, 0x00, 0x00, 0x00, 0x18, 0x24, 0x24,
                      0x18, 0x24, 0x24, 0x18, 0x00, 0x00, 0x18, 0x24,
                      0x42, 0x42, 0x42, 0x24, 0x18, 0x00, 0x00, 0x3e,
                      0x20, 0x3c, 0x02, 0x02, 0x22, 0x1c, 0x00, 0x00,
                      0x08, 0x18, 0x38, 0x18, 0x18, 0x18, 0x7e, 0x00,
                      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
unsigned char speed=20;             // 约0.32s
void delay1ms(unsigned char);       // 声明延迟函数
//============== 主 程 序 ==================
main()                              // 主程序开始
{  unsigned char i,j,k,scan;        // 声明变数
   while(1)                         // 无穷尽循环
   {  for(i=0;i<41;i++)             // 显示40个视窗
      {  for(k=0;k<speed;k++)       // 每个视窗重复执行k次
         {  scan=1;                 // 初始扫描信号
            for(j=0;j<8;j++)        // 扫描一周
```

```
            {   ROWP=0XFF;  // LED OFF
                P1=~scan;       // 输出扫描信号
                ROWP=~d[i+j]; // 输出显示信号
                delay1ms(2);    // 延迟2ms
                scan<<=1;}      // 下一个扫描信号
            }                   // 完成扫描一周
        }                       // 完成k次
    }                           // while叙述结束
}                               // 主程序结束
//============== 延迟函数 ==================
void delay1ms(unsigned char x)
{ unsigned char i,j;            // 声明变数
  for(i=0;i<x;i++)              // 外循环
     for(j=0;j<120;j++);        // 内循环
}                               // 延迟函数结束
```

8×8 LED阵列上卷实验（ch12-6.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图12-20连接线路，使用在线仿真器，载入新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

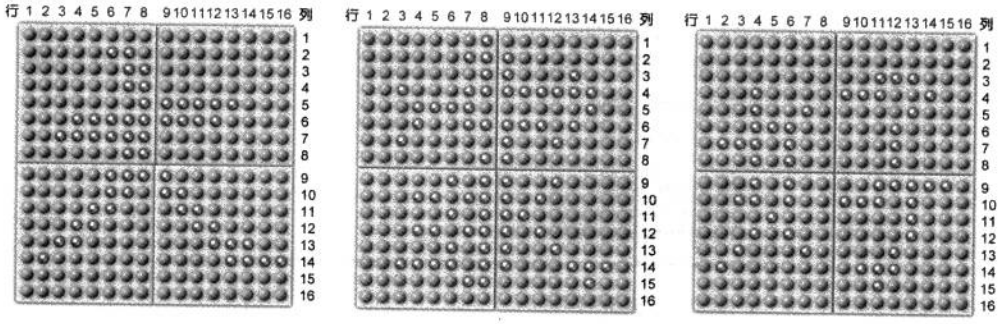
1. 在本实验里，文字呈现动态上移，请将它修改为动态下移？
2. 根据本实验的电路，请编写一个电话号码的动态上移？

12.5.7 16×16LED阵列显示实例演练

实验要点

如图12-27所示，其中由4个8×8共阳型LED阵列组成16×16阵列，其扫描信号连

接到8051的P1.0~P1.3，显示信号连接到8051的Port 2；另外，锁存信号分别接到P3.0及P3.1。在本单元里将由这个LED阵列静态显示“大家好”，约每0.48s换一个字。如图12-27所示为“大家好”的编码。



位数	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
大Hi	0x00	0x00	0x40	0x60	0x60	0x62	0xee	0xfc	0x30	0x30	0x30	0x30	0x10	0x00	0x00	0x00
大Lo	0x00	0x20	0x10	0x18	0x0c	0x07	0x03	0x01	0x03	0x06	0xc0	0x18	0x30	0x30	0x20	0x20
家Hi	0x00	0x00	0x48	0x30	0x10	0x10	0x3a	0xaf	0xee	0x28	0x28	0x48	0x2c	0x18	0x00	0x00
家Lo	0x00	0x00	0x40	0x2a	0x2a	0x35	0x4a	0x7f	0x3f	0x04	0x0a	0x11	0x20	0x60	0x20	0x00
好Hi	0x00	0x40	0x40	0xf8	0x20	0xe0	0x10	0x00	0x08	0x08	0x0c	0xe4	0x14	0x08	0x00	0x00
好Lo	0x00	0x20	0x12	0x0b	0x04	0x0b	0x10	0x00	0x02	0x22	0x63	0x31	0x0f	0x010	x01	0x00

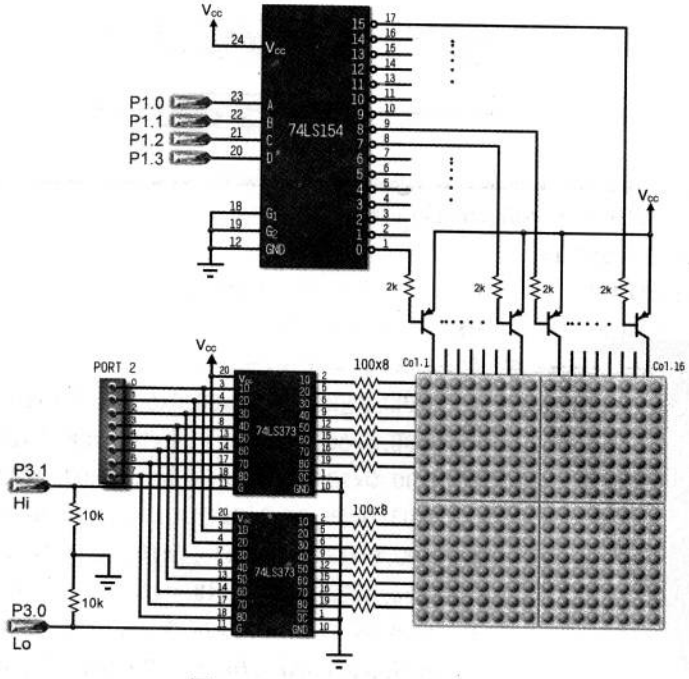
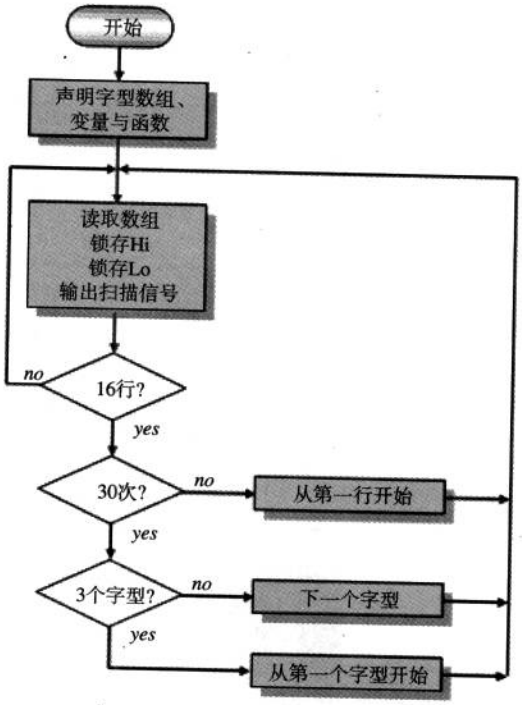


图12-27 16×16实验电路图

流程图与程序设计

16×16阵列的扫描方式必须配合锁存信号（正脉冲）。例如要显示第一条扫描线，则先送入上面部分的显示信号，再对上面部分的锁存器送一个正脉冲，即可将该信号锁在该锁存器里而不受影响；紧接着，下面部分的显示信号再对下面部分的锁存器送一个正脉冲，即可将该信号锁在该锁存器里而不受影响。最后送出该行的扫描信号，即可显示该行的16个LED。




```
/*16 x 16 LED阵列实验(ch12-7.c)*/
#include <reg51.h>
#define ROWP          P2      // 输出列接至P2
#define COLP          P1      // 扫描行接至P1
//============== 字 型 ================
unsigned char d[][][]={   {   {0x00, 0x00, 0x40, 0x60, 0x60, 0x62, 0xee, 0xfc, // 大
                              0x30, 0x30, 0x30, 0x30, 0x10, 0x00, 0x00, 0x00},
                          {   0x00, 0x20, 0x10, 0x18, 0x0c, 0x07, 0x03, 0x01,
                              0x03, 0x06, 0xc0, 0x18, 0x30, 0x30, 0x20, 0x20}},
                          {   0x00, 0x00, 0x48, 0x30, 0x10, 0x10, 0x3a, 0xaf,// 家
                              0xee, 0x28, 0x28, 0x48, 0x2c, 0x18, 0x00, 0x00},
                          {   0x00, 0x00, 0x40, 0x2a, 0x2a, 0x35, 0x4a, 0x7f,
                              0x3f, 0x04, 0x0a, 0x11, 0x20, 0x60, 0x20, 0x00}},
                          {   0x00, 0x40, 0x40, 0xf8, 0x20, 0xe0, 0x10, 0x00, // 好
```

```
                              0x08, 0x08, 0x0c, 0xe4, 0x14, 0x08, 0x00, 0x00},
                          {   0x00, 0x20, 0x12, 0x0b, 0x04, 0x0b, 0x10, 0x00,
                              0x02, 0x22, 0x63, 0x31, 0x0f, 0x01, 0x01, 0x00} }};
unsigned char speed=30;           // 约0.48s
sbit Hi = P3^1;                   // 声明Hi锁存信号位置
sbit Lo = P3^0;                   // 声明Lo锁存信号位置
void delay1ms(unsigned char);     // 声明延迟函数
//============== 主 程 序 ==================
main()                            // 主程序开始
{ unsigned char i,j,k,l,scan;     // 声明变数
  while(1)                        // 无穷尽循环
  {  Hi=0; Lo=0;                  // 关闭锁存器
     for(i=0;i<3;i++)             // 三个字
       for(k=0;k<speed;k++)       // 重复k次
       {  scan=0;                 // 初始扫描信号
          for(j=0;j<16;j++)       // 内循环
          {  P2=~d[i][0][j];      // 输出Hi部分
             Hi=1;                // 开启Hi锁存器
             for(l=0;l<6;l++);    // 延迟
             Hi=0;                // 关闭Lo锁存器
             P2=~d[i][1][j];      // 输出Lo部分
             Lo=1;                // 开启Lo锁存器
             for(l=0;l<6;l++);    // 延迟
             Lo=0;                // 关闭Lo锁存器
             P1=scan;             // 输出扫描信号
             delay1ms(1);         // 延迟1ms
             scan++;              // 下一个扫描信号
          }                       // 完成扫描一个字型
       }                          // 完成重复k次
  }                               // while结束
}                                 // 主程序结束
//============== 延迟函数 ==================
void delay1ms(unsigned char x)
{ int i,j;                        // 声明变数
  for(i=0;i<x;i++)                // 外循环
    for(j=0;j<120;j++);           // 内循环
}                                 // 延迟函数结束
```

16×16 LED阵列实验（ch12-7.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查源程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图12-28连接线路，使用在线仿真器，载入新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

1. 请将本实验里所显示的字型改成反白显示？

2. 请将在本实验里静态显示的方式改成左移的动态显示？

12.6 实时练习



在本章中探讨了LED阵列的结构与应用，包括如何应用8051驱动LED阵列的显示。在此请试着回答下列问题，以确认对于此部分的掌握程度。

选择题

（ ）1. 对于8×8 LED阵列而言，其中的LED个数及引脚数各为多少？
（A）64、16 （B）16、16 （C）64、12 （D）32、12

（ ）2. 在共阳极型8×8 LED阵列里，其阳极如何连接？
（A）各列阳极连接到列引脚 （B）各行阳极连接到行引脚
（C）各列阳极连接到行引脚 （D）各行阳极连接到列引脚

（ ）3. 同上题，其阴极如何连接？
（A）各列阴极连接到列引脚 （B）各行阴极连接到行引脚
（C）各列阴极连接到行引脚 （D）各行阴极连接到列引脚

（ ）4. 双色8×8 LED阵列的行引脚与列引脚数为多少？
（A）12、12 （B）8、16 （C）16、8 （D）24、8

（ ）5. “5×8 LED阵列”指的是何种LED阵列？
（A）5行8列的LED阵列 （B）5mm的8×8 LED阵列
（C）8行5列的LED阵列 （D）8mm的5×5 LED阵列

（ ）6. 通常8×8 LED阵列的驱动方式是什么？
（A）直接驱动 （B）扫描驱动 （C）双向驱动 （D）以上皆非

（ ）7. 对于*m*行*n*列的LED阵列而言，其扫描的工作周期是多少时比较不会感觉闪烁？
（A）16ms/*m* （B）16ms/*n* （C）64ms/*m* （D）16ms/*n*

（ ）8. 若要采用两个8位的输入/输出口驱动16×16 LED阵列，必须使用哪种辅助器件？
（A）译码器 （B）多工器 （C）解多工器 （D）锁存器

（ ）9. 下列哪个器件可提供1对16的解码功能？
（A）74138 （B）74139 （C）74154 （D）74373

（ ）10. 在16×16 LED阵列驱动电路里，通常会使用1对16译码器，以作为何种用途？
（A）产生扫描信号 （B）锁存扫描信号
（C）锁存显示信号 （D）放大驱动电流

问答题

1. 共阳型LED阵列是指其每行LED的阳极都连接在一起，还是每列LED的阳极都连接在一起？

2. 5×7 LED阵列是指该LED阵列是由多少行多少列所组成？

3. LED阵列的行引脚所负担的电流比较多，还是列引脚所负担的电流比较多？

4. 74LS373的功能是什么？其I_{OL}约为多少？

5. 74LS154的功能是什么？其G_1、G_2引脚的功能是什么？

6. 试述LED阵列显示的动作原理。

7. 若要LED阵列进行左移/右移显示，应如何处理？

8. 若要LED阵列进行上移/下移显示，应如何处理？

9. 多色LED阵列（如MM12884AG）最多可以显示多少个颜色？

10. 若要用4个8×8 LED阵列组成16×16的LED显示幕，其电路应如何处理？

() 9. 如果8×8 LED点阵显示屏的行线为阳极，则列线为阴极。

() 10. 8×8 LED点阵显示屏的行线为阴极，则列线为阳极。

二、选择题

(A) 发光二极管 (B) 数码管
(C) 点阵显示屏 (D) 数字显示屏

() 1. 点阵显示屏的英文缩写是（ ）。

(A) LED (B) LCD (C) 7SEG (D) DOT

() 2. 8×8 LED点阵显示屏的行线为阳极，则列线为（ ）。

(A) 阴极 (B) 阳极 (C) 悬空 (D) 接地

() 3. 8×8 LED点阵显示屏的行线为阴极，则列线为（ ）。

(A) 阴极 (B) 阳极 (C) 悬空 (D) 接地

() 4. 对于8×8 LED点阵显示屏，其行线通常接（ ）。

(A) 74138 (B) 74139 (C) 74154 (D) 74273

() 5. 对于8×8 LED点阵显示屏，其列线通常接（ ）。

(A) 74138 (B) 74139 (C) 74154 (D) 74273

() 6. 8×8 LED点阵显示屏的行线为阳极，则列线为阴极，若要点亮第一行第一列的LED，则应（ ）。

(A) 行线接高电平，列线接低电平 (B) 行线接低电平，列线接高电平
(C) 行线接高电平，列线悬空 (D) 行线接低电平，列线悬空

() 7. 8×8 LED点阵显示屏的行线为阴极，则列线为阳极，若要点亮第一行第一列的LED，则应（ ）。

(A) 行线接高电平，列线接低电平 (B) 行线接低电平，列线接高电平
(C) 行线接高电平，列线悬空 (D) 行线接低电平，列线悬空

() 8. 8×8 LED点阵显示屏的行线为阳极，则列线为阴极，若要点亮第一行第一列的LED，则应（ ）。

(A) 行线接高电平，列线接低电平 (B) 行线接低电平，列线接高电平
(C) 行线接高电平，列线悬空 (D) 行线接低电平，列线悬空

() 9. 8×8 LED点阵显示屏的行线为阴极，则列线为阳极，若要点亮第一行第一列的LED，则应（ ）。

(A) 行线接高电平，列线接低电平 (B) 行线接低电平，列线接高电平
(C) 行线接高电平，列线悬空 (D) 行线接低电平，列线悬空

() 10. 8×8 LED点阵显示屏的行线为阴极，则列线为阳极，若要点亮第一行第一列的LED，则应（ ）。

(A) 行线接高电平，列线接低电平 (B) 行线接低电平，列线接高电平
(C) 行线接高电平，列线悬空 (D) 行线接低电平，列线悬空

三、简答题

1. 点阵型LED的行线通常接阳极还是阴极？

2. 8×8 LED点阵显示屏的行线为阳极，则列线为阴极，若要点亮第一行第一列的LED，则应如何操作？

3. LED点阵显示屏的行线为阴极，则列线为阳极，若要点亮第一行第一列的LED，则应如何操作？

4. 74LS273的作用是什么？

5. 74LS154的作用是什么？

6. 点亮LED点阵显示屏的原理是什么？

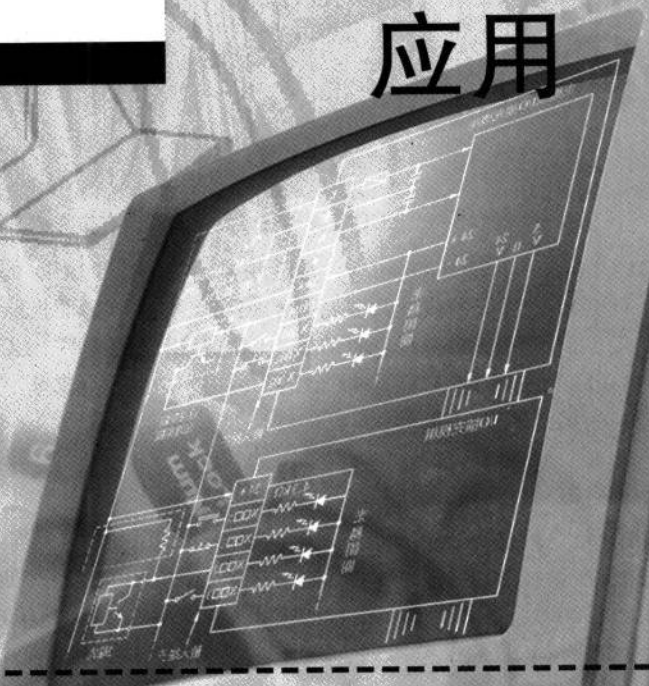
7. 点亮LED点阵显示屏的行线为阳极，则列线为阴极，若要点亮第一行第一列的LED，则应如何操作？

8. 点亮LED点阵显示屏的行线为阴极，则列线为阳极，若要点亮第一行第一列的LED，则应如何操作？

9. 点亮LED点阵显示屏的行线为阳极，则列线为阴极，若要点亮第一行第一列的LED，则应如何操作？

10. 点亮8×8 LED点阵显示屏的行线为阴极，则列线为阳极，若要点亮第一行第一列的LED，则应如何操作？

第13章 LCD模块的应用



本章内容丰富，主要包括以下两部分。

硬件部分：

LCD模块的结构及其与8051的接口。

LCD模块的指令、编码与其应用。

介绍中文LCD模块与其应用。

程序与实践部分：

基本LCD模块的应用、自建字型等。

13.1 认识LCD模块

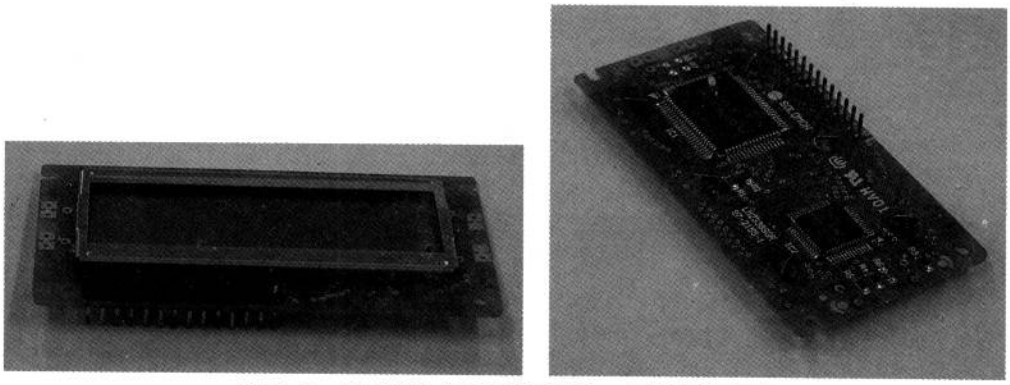


图13-1 LCD模块（左边为正面图，右边为背面图）

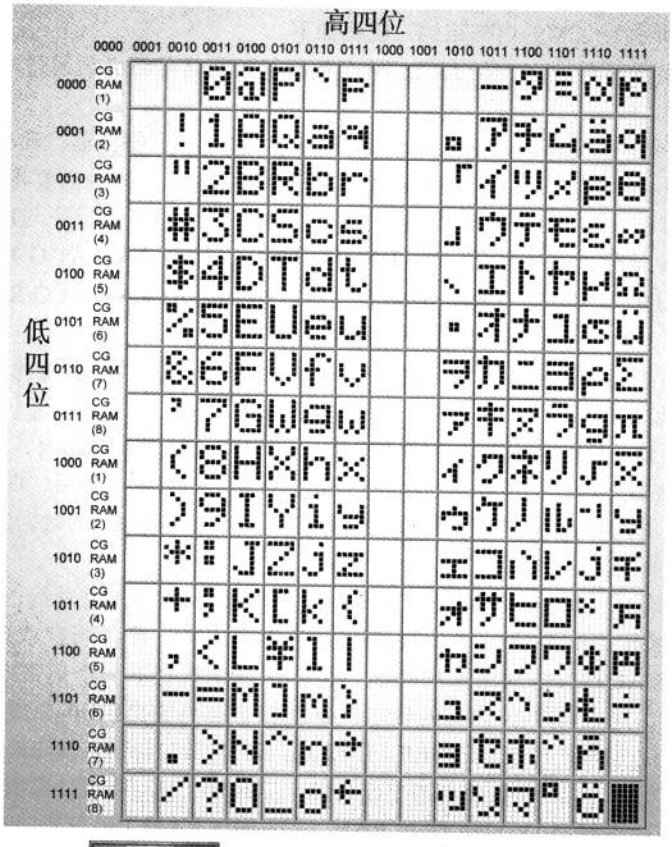
LCD（Liquid Crystal Display）是液晶显示器，由于LCD 的控制必须使用专用的驱动电路，而且LCD面板的接线需要特殊的技巧，加上LCD面板结构较脆弱，因此一般不会单独使用，而是将LCD面板、驱动与控制电路组合而成一个LCD模块（Liquid Crystal Display Moulde，简称为**LCM**）。LCM是一种很省电的显示装置，常被应用于数字或微型计算机控制的系统，作为简易的人机接口，如图13-1所示。

LCM基本数据

LCM的种类繁多，而在学校与训练单位所采用的LCM大都是用日立公司的控制器（HD44780）所组成的LCM，其内部结构如图13-2所示，其特性如下。

- 内含80KB数据显示存储器（Data Display RAM，简称**DD RAM**），可显示16字×1列、20字×1列、16字×2列、20字×2列、40字×2列等模式。
- 内含字型产生器（Character Generate ROM，简称**CG ROM**），可产生160个5×7字型，如表13-1所示。
- 自建字型产生器（Character Generate RAM，简称**CG RAM**），用户可以自建8个5×7字型。

表13-1



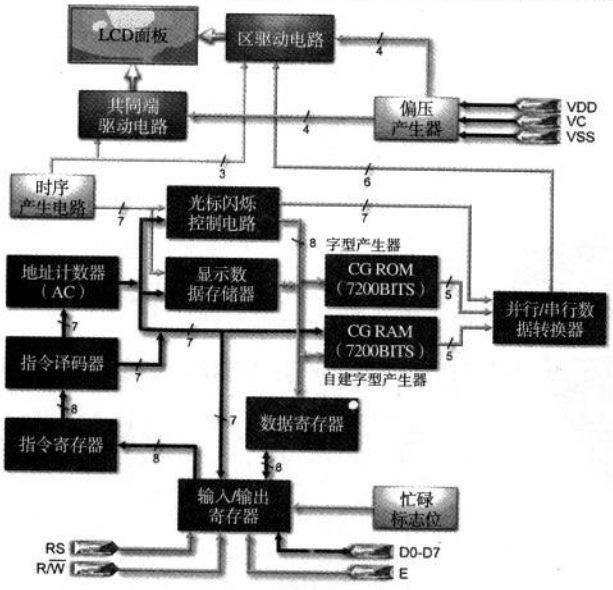


图13-2 HD44780 LCM内部结构

LCM内部结构

如图13-2所示为LCM的内部结构，说明如下。

- 输入/输出寄存器是LCM的大门，所有数据与控制信号都必须通过本单元才可以进出LCM。
- 指令寄存器（Instruction Register，简称**IR**）是一个8位寄存器，其功能是存放微处理器所送入的LCM指令、DD RAM或CG RAM的地址。当我们要将数据输入到DD RAM或CG RAM时，首先将数据放入数据寄存器，再把指令与DD RAM或CG RAM的地址放入本寄存器，即可将该数据输入到DD RAM或CG RAM。同样地，若要读取DD RAM或CG RAM的数据，则将指令与DD RAM或CG RAM的地址放入本寄存器，即可在数据寄存器中取得该地址的数据。
- 指令译码器的功能是将指令寄存器里的指令译码，以获得所要操作DD RAM或CG RAM的地址。
- 数据寄存器（Data Register，简称**DR**）连接LCM内部数据总线，DD RAM或CG RAM的数据存取，都需要通过本寄存器。当CPU读取DR内容后，DR将自动加载下一个地址的内容。因此，若要连续读取DD RAM或CG RAM的数据，只要指定其起始的地址即可。
- 地址计数器（Address Counter，简称**AC**）连接LCM内部地址总线，DD RAM或CG RAM的操作都需要通过本计数器所提供的地址来寻址。当存取DD RAM或CG RAM时，AC具有自动增加的功能，也就是自动指到下一个记忆地址。当RS=0，R/W=1时，进入读取AC内容的状态，AC里的数据将输出到D0~D7数据总线上。
- 忙碌标志位（Busy Flag，简称**BF**）用以表示LCM当时的状态。若BF=1，则表示LCM处于忙碌状态，无法接收外部指令或数据；若BF=0，则可接收外部指令或数据。
- 显示数据存储器（Display Data RAM，简称**DD RAM**）映象所要显示的数据，是LCM的主战场。实际上，在本存储器里存放的是所要显示数据的ASCII码，作用是把该ASCII码作为地址，到CG ROM或HCG ROM里能找到该字型的显示编码。DD RAM的存储器地址分为4行，第1行的地址为0x80～0x8f、第2行的地址为0x90～0x9f、第3行的地址为0xa0～0xaf、第4行的地址为0xb0～0xbf，其中第1行与第2行直接映对到LCD面板，如图13-3所示。

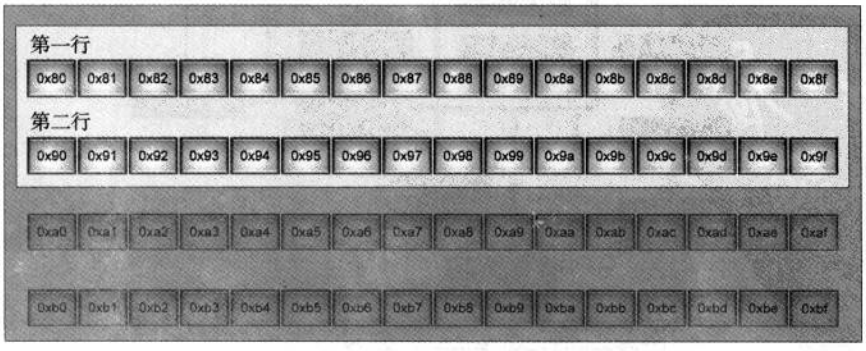


图13-3 DDRAM存储器地址

而第3行与第4行空的存储器位置在LCD面板上没有实际对应的显示区，所以就当它们不存在。

- 字型产生器（Character Generate ROM，简称**CG ROM**）是一个只读存储器，其中包括所有预置的显示数据的编码，而这个编码表就是ASCII编码。
- 自建字型产生器（Character Generate RAM，简称**CG RAM**）是一个随机存取存储器，其功能是提供存放用户所建立的字型样板（pattern），最多可自建8个字型。如图13-4所示，分别为自建的“☺”及“±”。

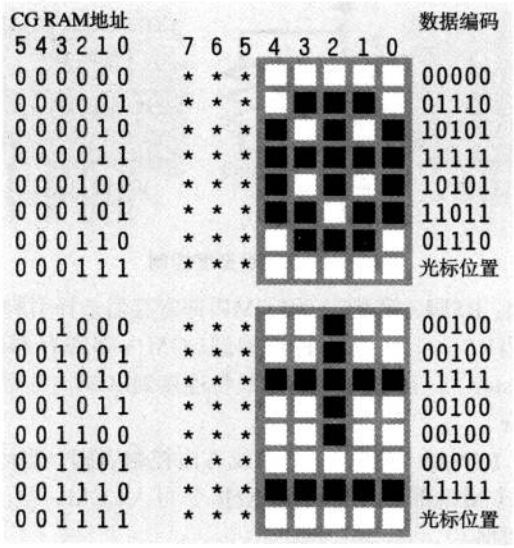


图13-4 自建字型

每个字型都由8组数据编码所构成，每个数据编码仅用到前5个位（4~0）。若要显示则在该位置标示“1”，在不要显示的位置标示“0”，最后一组编码通常是空白的，留给光标使用。而这8组数据编码在CG RAM的地址是以其低3位来编列，分别为000到111。CG RAM提供8个字型的地址，总共8×8个字节存储器空间，高3位000代表第一个自建字型，001代表第2个自建字型……，111代表第8个自建字型。

- 串行/并行数据转换器（parallel-serial converter）的功能是将从CG RAM或CG ROM所取出的并行显示数据转换成串行数据，以提供驱动电路来驱动LCD面板。
- 光标闪烁控制电路（cursor/blink controller）的功能是控制光标以及闪烁字型的产生。
- 时序产生电路（timing generator）的功能是产生LCM所需要的时钟脉冲。
- 偏压产生电路（bias voltage generator）的功能是提供驱动LCD面板所需要的偏压。
- 共同端驱动电路（common driver）的功能是提供LCD面板共同端的扫描信号。
- 区段驱动电路（segment dirver）的功能是提供LCD面板的显示信号。
- LCD面板（LCD panel）是一种点矩阵式液晶显示器。

LCM的引脚

如图13-1所示，LCM包括14个引脚，说明如下。

- 电源引脚：第2脚VDD为电源引脚，连接+5V即可。而第1脚VSS为接地引脚。另外，第3脚Vo为显示屏明亮度调整引脚，此引脚的电压越低，则显示屏明亮度越高。我们可以利用一个10kΩ的可变电阻（或半固定电阻）连接成明亮度调整电路，如图13-5所示。若不想太麻烦，则将Vo与VSS连接后接地即可。

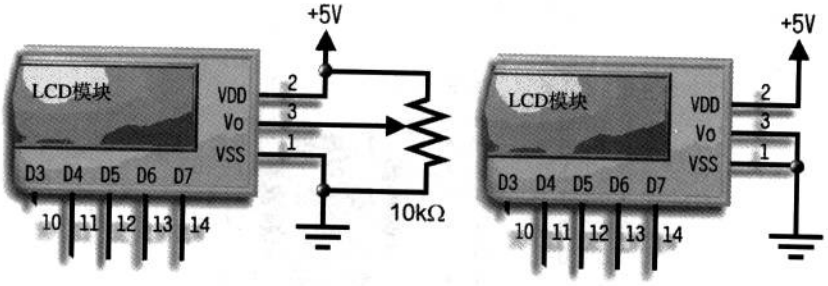


图13-5 明亮度控制

- 寄存器选择引脚：RS脚（第4脚）为LCM内部寄存器选择引脚（即Register Selector，简称为**RS**）。当RS=0时，总线将连接到LCM内部寄存器的指令寄存器IR（即Instruction Register）；当RS=1时，总线将连接到LCM内部寄存器的数据寄存器DR（即Data Register）。
- 读写控制引脚：R/W脚（第5脚）为总线方向控制引脚，当R/W=0时，总线将由微处理器输入到LCM内部，以进行数据/指令写入LCM；当 R/$\overline{W}$=1时，总线将由LCM内部读取数据。
- 启用引脚：E脚（第6脚）为LCM的启用信号，此为负边缘触发式引脚。
- 数据总线引脚：第7脚~第14脚为数据总线引脚，即D0~D7。

LCM的封装

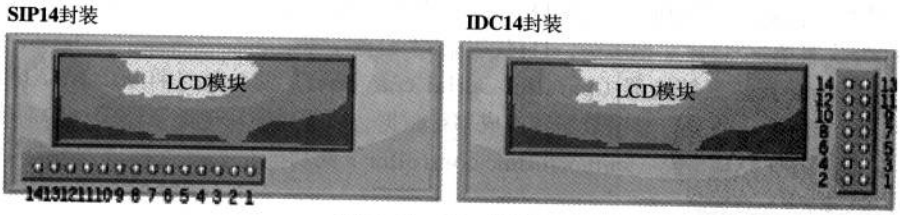


图13-6 LCM的封装

常见的LCM引脚封装有两种，如图13-6所示，第一种采用单排引脚封装（SIP14），第二种采用双排引脚封装（IDC14）。至于引脚的实际位置，不同的厂牌、型号的器件会有所不同，使用之前必须详细阅读LCM的使用说明。

13.2 中文LCD模块

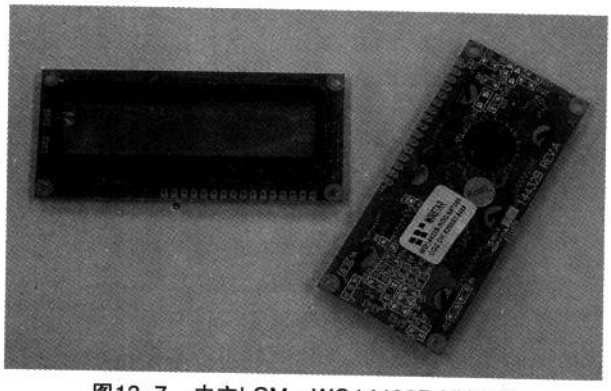


图13-7 中文LCM－WG14432B-YYH-N

一般的LCM都会预留8×8个字节的存储器空间（CG RAM），让用户自建图形或字型。8×8个字节可建立8个5×8的图案，若要勉强制作中文字型，则需要两个5×8的图案才能构成一个中文字，也就是最多可以自建4个中文字，显然这不是个好主意！而中文市场在日渐增长，LCM厂商绝不会坐视不管。所以，近年来中文LCM大行其道。尽管如此，中文LCM与一般非中文LCM的差别不大，只是中文LCM多出了中文编码的ROM，外表上很难分辨其差异，而驱动的指令、外部引脚等并无不同！以下将介绍华凌光电股份有限公司（www.winstar.com.tw）生产的型号为**WG12864J-YYH-N**的中文LCM模块。

型号的编号方式

W G 1 4 4 3 2 B－Y Y H－N
① ② ③ ④ ⑤ ⑥ ⑦ ⑧

项目	功 能	说 明
①	厂牌	W代表Winstar Display 公司所生产的产品
②	显示种类	H代表文字形式　　G代表图形形式
③	显示字型	144×32点阵
④	模块序号	器件编号
⑤	背光形式	N代表没有背光 B代表EL，蓝灰色　　A代表LED，琥珀色 D代表EL，绿色　　R代表LED，红色 W代表EL，白色　　O代表LED，橙色 F代表CCFL，白色　　G代表LED，绿色 Y代表LED，黄绿色

续表

项目	功 能	说 明
⑥	LCD模式	B代表TN正，灰色　　F代表FSTN正 N代表TN负 T代表FSTN负 G代表STN正，灰色 Y代表STN正，黄灰色 M代表STN负，蓝色
⑦	LCD偏光镜 温度范围 展示方向	A代表反射式，一般温度范围，6点钟方向 D代表反射式，一般温度范围，12点钟方向 G代表反射式，宽温度范围，6点钟方向 J代表反射式，宽温度范围，12点钟方向 B代表透射式，一般温度范围，6点钟方向 E代表透射式，一般温度范围，12点钟方向 H代表透射式，宽温度范围，6点钟方向 K代表透射式，宽温度范围，12点钟方向 C代表透射式，一般温度范围，6点钟方向 F代表透射式，一般温度范围，12点钟方向 I代表透射式，宽温度范围，6点钟方向 L代表透射式，宽温度范围，12点钟方向
⑧	特殊码	N代表不需要负电压

引脚表

引脚号码	引脚名称	电平	说 明
1	VSS	0V	接地
2	VDD	5.0V	LCD模块逻辑电路电源（+5V）
3	Vo	–	LCD电源
4	RS	H/L	RS=1，处理数据 RS=0，处理指令
5	R/$\overline{W}$	H/L	R/W=1，读取LCM（MPU←LCM） R/W=0，写入LCM（MPU→LCM）
6	E	H/L	启用信号
7	DB0	H/L	总线
8	DB1	H/L	总线
9	DB2	H/L	总线
10	DB3	H/L	总线
11	DB4	H/L	总线
12	DB5	H/L	总线
13	DB6	H/L	总线
14	DB7	H/L	总线
15	A	–	背光LED的正端
16	K	–	背光LED的负端

内部结构

如图13-8所示，WG12864J-YYH-N系列采用LCD驱动芯片大厂硅创电子股份有限公司（www.sitronix.com.tw）生产的中文LCD驱动控制器 ST7920，而不是前面所介绍的HD44780，但其指令与操作方式几乎完全一样。

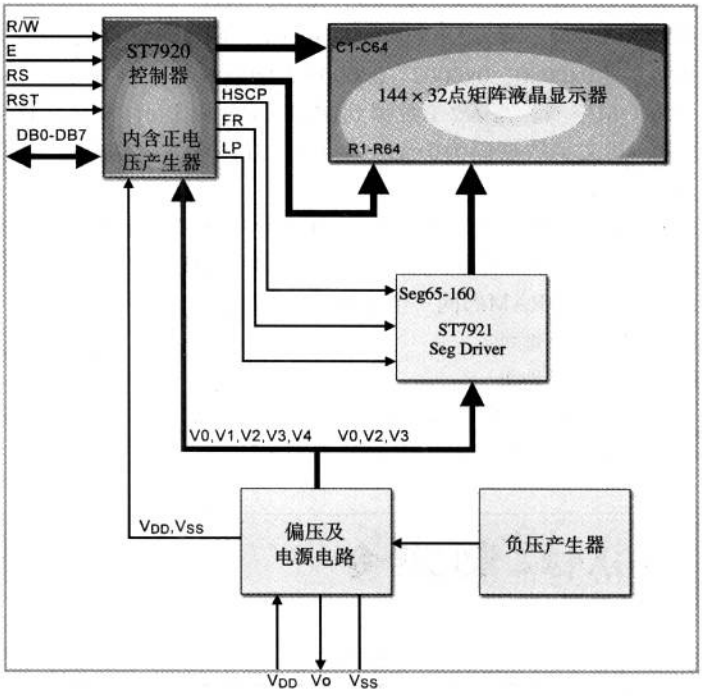


图13-8 中文LCM－WG14432B-YYH-N的内部结构

字型产生器

ST7920控制器提供具有8192个16×16字型（可为中文字）及126个8×16字型（英文字母与数字）的存储器（ROM），这样将可以支持多语文的应用，例如中文与英文并行。对于连续的两个字节，可用来指定一个16×16字型或两个8×16字型，而8×16字型就是通常所说的半高字型（half-height characters），所指定的字型码将被写入DDRAM，同时从CGROM或HCGROM对应到其字型，即可显示它。

- 自建字型产生器：ST7920控制器提供4组用户定义字型（16×16字型）的存储器（即**CGRAM**），而其显示方式与前面讲的对应到CGROM或HCGROM的方式一样。
- 自建图标产生器：ST7920控制器提供240个图标（Icon），包括15组IRAM（即Icon RAM）的地址，每组IRAM地址包含16位数据，而其数据安排的顺序是高8位（D15~D8）先，低8位（D7~D0）后。
- 显示数据存储器：DDRAM（Display Data RAM）是指显示数据存储器，ST7920控制器里的DDRAM可以储存16个16×16字型（4列）或32个8×16字型（4列），当然，在**WG12864J-YYH-N**的LCD面板里，最多只能显示两列。文字数据码储存在

DDRAM里，而对应到CGROM、HCGROM及CGRAM。ST7920能够显示HCGROM的半高字型、CGRAM里用户自行定义的字型以及CGROM里的16×16全高字型，说明如下。

- 显示HCGROM半高字型：写入2 B数据到DDRAM，则显示2个8×16字型，每个byte代表一个字型；而其文字数据码为0x02~0x7f。
- 显示CGRAM字型：写入2 B数据到DDRAM，则显示1个16×16字型；而其文字数据码只能为0x0000、0x0002、0x0004及0x0006。
- 显示CGROM字型：写入2 B数据到DDRAM，则显示1个16×16字型。若采用big-5码，其文字数据码为0xa140～0xd75f；若采用GB码，其文字数据码为0xa1a0～0xf7ff。

• 显示图案存储器：GDRAM（即Graphic Display RAM）提供64×256位、采用位对应存储器空间。GDRAM的地址是由连续的两B所组成，分为垂直地址与水平地址，两个B数据将写入GDRAM的同一个地址里，其步骤如下。

1. 设定GDRAM的垂直地址。
2. 设定GDRAM的水平地址。
3. 将D15～D8写入GDRAM（第1个字节）。
4. 将D7～D0写入GDRAM（第2个字节）。

13.3 LCM控制指令



表13-2

功　能	控制线		总线								执行时间
	RS	R/$\overline{W}$	D7	D6	D5	D4	D3	D2	D1	D0	
清除显示屏	0	0	0	0	0	0	0	0	0	1	1.6ms
清除显示屏，并把光标移至左上角											
光标归位	0	0	0	0	0	0	0	0	1	×	72μs
光标移至左上角，显示内容不变											
设定输入模式	0	0	0	0	0	0	0	1	I/O	S	72μs
I/D=1：地址递增，I/D=1：地址递减 S=1：开启显示屏，S=0：关闭显示屏											
开关显示屏	0	0	0	0	0	0	1	D	C	B	72μs
D=1开启显示屏，D=0关闭显示屏 C=1开启光标，C=0关闭光标 B=1光标所在位置的字符闪烁、B=0光标所在位置的字符不闪烁											
移位方式	0	0	0	0	0	1	S/C	R/L	×	×	72μs
S/C=1显示屏移位，S/C=0光标右移 R/L=1向右移，R/L=0向左移											

续表

功 能	控制线		总线								执行时间
	RS	R/$\overline{W}$	D7	D6	D5	D4	D3	D2	D1	D0	
功能设定	0	0	0	0	1	DL	×	RE	×	×	72μs
DL=1数据长度为8位，DL=0数据长度为4位 RE=1采用扩展指令，RE=0采用一般指令，请参阅随书光盘里的ST7920的data sheet，在此不探讨扩展指令											
CGRAM寻址	0	0	0	1	CG RAM地址						72μs
将所要操作的CG RAM地址放入地址计数器											
DDRAM寻址	0	0	1		DD RAM地址						72μs
将所要操作的DD RAM地址放入地址计数器											
读取BF与AC	0	1	BF		地址计数器内容						0μs
读取地址计数器，并查询LCM是否忙碌 BF=1表示LCM忙碌，BF=0表示LCM可接收指令或数据											
写入数据	1	0			所要写入的数据						72μs
将数据写入内部存储器（GDRAM、IRAM、CG RAM和DD RAM）											
读取数据	1	1			所要读取的数据						72μs
读取内部存储器（GDRAM、IRAM、CG RAM和DD RAM）的数据											

执行时间是根据ST7920在540kHz下的执行时间得出的

LCM的控制指令只有11个，如表13-2所示，在本单元里，将依次介绍这11个指令。

清除显示屏（CLEAN DISPLAY）

RS	R/W
0	0

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	1

RS=0，R/$\overline{W}$=0是执行指令写入的操作，而数据总线上的指令为00000001（即0x01），其动作如下。

1. 让显示屏变成空白，LCM将会把DD RAM全部填入20H（即空白）。
2. 将光标移至左上角（HOME）。
3. 使地址计数器（AC）归零。
4. 整个执行时间需要1.6ms。

光标归位

RS	R/W
0	0

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	1	*

RS=0， R/$\overline{W}$=0是执行指令写入的操作，而数据总线上的指令为0000001*（即0x02或0x03），其中的“*”代表可为0或1，而其动如下。

1. 将光标移至左上角（HOME），但DD RAM的内容不变。
2. 使地址计数器（AC）归零。
3. 整个执行时间需要72μs。

设定输入模式

RS	R/W
0	0

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	1	I/D	S

RS=0， R/$\overline{W}$=0是执行指令写入的操作，而数据总线上的指令为000001 I/D S，其中的I/D与S位如下：

I/D	S	功能
0	0	显示的字符不动，光标左移，AC-1
0	1	显示的字符右移，光标不动，AC不变
1	0	显示的字符不动，光标右移，AC+1
1	1	显示的字符左移，光标不动，AC不变

设定显示屏

RS	R/W
0	0

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	1	D	C	B

RS=0， R/$\overline{W}$=0是执行指令写入的操作，而数据总线上的指令为00001 D C B，其中的D、C与B位如下。

1. D位显示屏控制开关，D=1时可开启显示屏，D=0时则关闭显示屏。
2. C位光标控制开关，C=1时可显示光标，C=0时则不显示光标。
3. B位字符闪烁控制开关，B=1时则光标所在的字符将闪烁，B=0时则光标所在的字符将不闪烁。
4. 整个执行时间需要72μs。

设定移位方式

RS	R/W
0	0

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	1	S/C	R/L	*	*

RS=0， R/$\overline{W}$=0是执行指令写入的操作，而数据总线上的指令为0001* S/C R/L，其中的“*”代表可为0或1，而S/C与R/L位如下：

S/C	R/L	功 能
0	0	光标左移，AC-1
0	1	光标右移，AC+1
1	0	整个显示屏左移
1	1	整个显示屏右移

功能设定

RS	R/W
0	0

D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	DL	*	RE	*	*

RS=0， R/$\overline{W}$=0是执行指令写入的操作，而数据总线上的指令为001 DL N F **，其中的“*”代表可为0或1，而DL、N与F位如下。

1. DL位为传送的数据长度设定，DL =1则采用8位方式的数据传送，DL =0则采用4位方式的数据传送，其中高4位先传送，再传送低4位。

2. RE位为扩展指令设定位，RE =1采用扩展指令，RE =0则采用一般指令。

3. 整个执行时间需要72μs。

CG RAM寻址

RS	R/W
0	0

D7	D6	D5	D4	D3	D2	D1	D0
0	1	A5	A4	A3	A2	A1	A0

RS=0， R/$\overline{W}$=0是执行指令写入的操作，而数据总线上的指令为01A5 A4 A3 A2 A1 A0，其中的A5 A4 A3 A2 A1 A0代表所要操作的CG RAM地址。紧接于本指令之后，即可将所要输入的数据，输入到这个地址。而整个执行时间需要72μs。

DD RAM寻址

RS	R/W
0	0

D7	D6	D5	D4	D3	D2	D1	D0
1	A6	A5	A4	A3	A2	A1	A0

RS=0， R/$\overline{W}$=0是执行指令写入的操作，而数据总线上的指令为1 A6 A5 A4 A3 A2 A1 A0，其中的A6 A5 A4 A3 A2 A1 A0代表所要操作的DD RAM地址。紧接于本指令之后，即可将所要输入的数据，输入到这个地址。而整个执行时间需要72μs。

读取BF与AC

RS	R/$\overline{W}$
0	1

D7	D6	D5	D4	D3	D2	D1	D0
BF	A6	A5	A4	A3	A2	A1	A0

RS=0，R/$\overline{W}$=1是执行读取的操作，这时候，LCM的忙碌标志位BF将放置在数据总线上的D7位，而LCM的地址计数器内容也将放置在数据总线上的D6~D0位，分别为A6 A5 A4 A3 A2 A1 A0。整个执行时间需要0μs。

数据写入

RS	R/$\overline{W}$
1	0

D7	D6	D5	D4	D3	D2	D1	D0
D7	D6	D5	D4	D3	D2	D1	D0

RS=1，R/$\overline{W}$=0是执行数据写入的操作，这时候，在数据总线上的数据将写入前一个指令所指定的DD RAM或CG RAM地址里。整个执行时间需要72μs。

读取数据

RS	R/$\overline{W}$
1	1

D7	D6	D5	D4	D3	D2	D1	D0
D7	D6	D5	D4	D3	D2	D1	D0

RS=1，R/$\overline{W}$=1是执行读取数据的操作，这时候，前一个指令所指定的DD RAM或CG RAM地址中的数据，将被放置在数据总线上。而读取数据之后，地址计数器将自动加1，指向下一个地址。整个执行时间需要72μs。

13.4 LCM的初始设定与常用函数



基本上，LCM是一个小系统，不管是HD44780还是ST79202都是微控制器。既然是微控制器，就有初始设定的问题，而LCM的数据模式有8位与4位（可利用DL位来设定），两种模式的初始设定有些差异，说明如下。

8位模式的初始设定

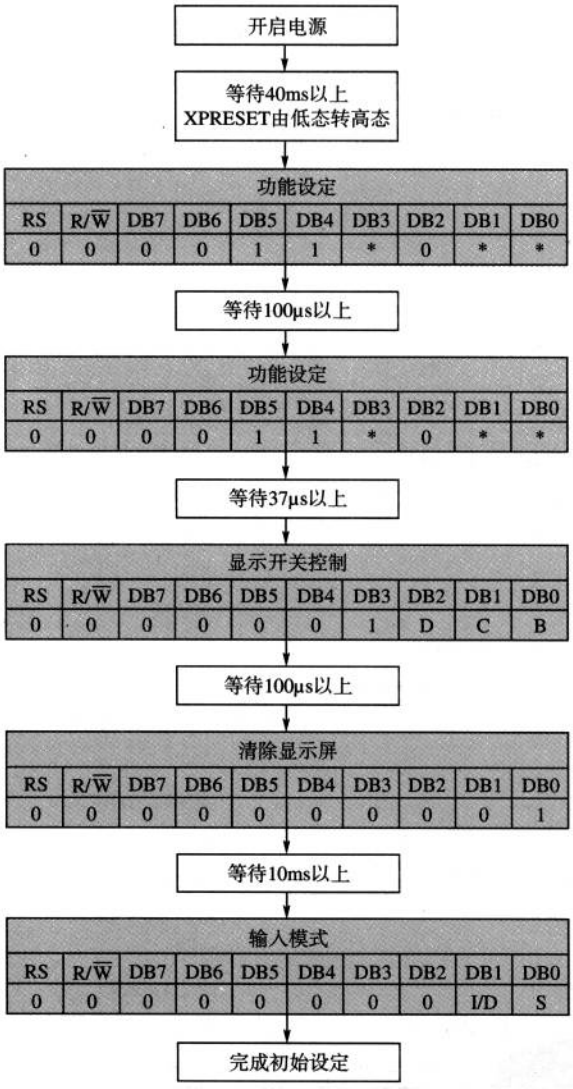


图13-9 8位模式的初始设定

根据上述初始设定的流程，若采用8位的数据传输模式（LCM与微控制器之间的数据总线为8位），而8051的Port 1连接总线，P3.0连接E、P3.1连接RW、P3.2连接RS，则在送入电源后，我们可以用下列指令进行初始设定：

```
sbit  en = P3^0;                          // 声明en的地址
sbit  RW = P3^1;                          // 声明RW的地址
sbit  RS = P3^2;                          // 声明RS的地址
//=========================================
main()
{ RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x30;                              // 设定功能
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
  RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x30;                              // 设定功能
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
  RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x08;                              // 关闭显示功能
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
  RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x01;                              // 清除显示屏
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
  RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x06;                              // 设定输入模式
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
```

4位模式的初始设定

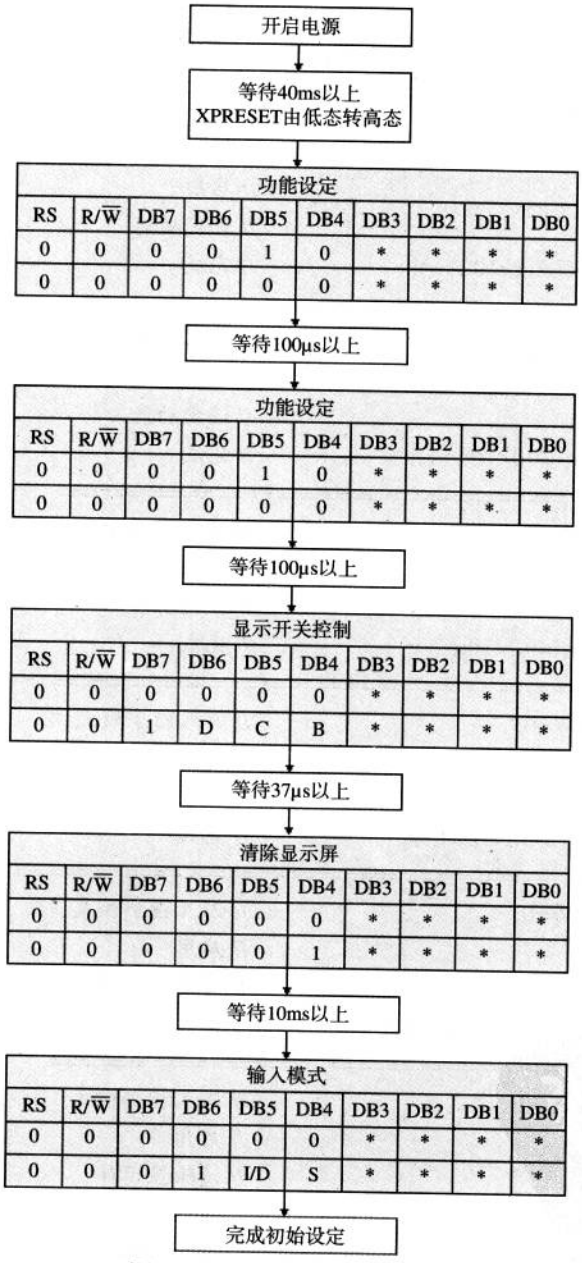


图13-10 4位模式的初始设定

根据上述初始设定的流程，若采用4位的数据传输模式（LCM与微控制器之间的数据总线为4位），很明显每个指令分两次送入LCM，在送入电源后，我们可以下列指令进行初始设定。

```
sbit  en = P3^0;                          // 声明en的地址
sbit  RW = P3^1;                          // 声明RW的地址
sbit  RS = P3^2;                          // 声明RS的地址
//=========================================
main()
{ RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x20;                              // 设定功能
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
  RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x00;                              // 设定功能
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
  RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x20;                              // 设定功能
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
  RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x00;                              // 关闭显示功能
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
  RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x80;                              // 关闭显示功能
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
  RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x00;                              // 清除显示屏
  en = 0; for(i=0;i<10;i++);              // 完成
//=========================================
  RS = 0; RW=0;                           // 写入指令模式
  en = 1;                                 // 启用
  P1 = 0x10;                              // 清除显示屏
```

```
  en = 0; for(i=0;i<10;i++);                    // 完成
//=============================================
  RS = 0; RW=0;                                 // 写入指令模式
  en = 1;                                       // 启用
  P1 = 0x00;                                    // 设定输入模式
  en = 0; for(i=0;i<10;i++);                    // 完成
//=============================================
  RS = 0; RW=0;                                 // 写入指令模式
  en = 1;                                       // 启用
  P1 = 0x06;                                    // 设定输入模式
  en = 0; for(i=0;i<10;i++);                    // 完成
//=============================================
```

● 写入指令函数

不管是8位还是4位的数据传输模式，都是一堆重复的动作，从上面的程序看起来，有点笨！我们可利用函数来执行写入指令的动作，以简化初始设定的程序，如下所示。

```
//====写入指令函数================================
void write_inst(char inst)
{ RS = 0; RW=0;                                 // 写入指令模式
  en = 1;                                       // 启用
  P1 = inst;                                    // 写入指令
  en = 0; for(i=0;i<10;i++);                    // 完成
}                                               // 函数结束
```

利用这个函数，则8位数据传输模式的初始设定简化，并写成一个初始设定的函数，而在初始设定的最后，将开启显示功能，如下所示。

```
//====初始设定函数(8位传输模式)==================
void init_LCM(void)
{ write_inst(0x30);                             // 设定功能
  write_inst(0x30);                             // 设定功能
  write_inst(0x08);                             // 关闭显示功能
  write_inst(0x01);                             // 清除显示屏
  write_inst(0x06);                             // 设定输入模式
  write_inst(0x0e);                             // 开启显示功能
}                                               // 函数结束
```

利用这个函数，则4位数据传输模式的初始设定函数，如下所示。

```
//====初始设定函数(4位传输模式)==================
void init_LCM(void)
{ write_inst(0x20);                    //设定功能
  write_inst(0x00);                    //设定功能
  write_inst(0x20);                    //设定功能
  write_inst(0x00);                    //设定功能
  write_inst(0x00);                    //关闭显示功能
  write_inst(0x08);                    //关闭显示功能
  write_inst(0x00);                    //清除显示屏
  write_inst(0x01);                    //清除显示屏
  write_inst(0x00);                    //设定输入模式
  write_inst(0x06);                    //设定输入模式
  write_inst(0x00);                    //开启显示功能
  write_inst(0x0e);                    //开启显示功能
}                                      //函数结束
```

兼容的初始设定

上述的初始设定参照了ST7920控制器的使用说明，主要是针对中文LCM。我们可将它改成与HD44780控制器兼容的初始设定，让它能同时用于ST7920控制器和HD44780控制器，如下所示。

```
//====初始设定函数(8位传输模式)==================
void init_LCM(void)
{       write_inst(0x30);              //设定功能
  write_inst(0x30);            //设定功能
  write_inst(0x30);            //设定功能
  write_inst(0x38);            //设定两列、5x7字型(HD44780)
  write_inst(0x08);            //关闭显示功能
  write_inst(0x01);            //清除显示屏
  write_inst(0x06);            //设定输入模式
  write_inst(0x0e);            //开启显示功能
}                              //函数结束
```

4位数据传输模式的初始设定函数，如下所示。

```
//====初始设定函数(4位传输模式)==================
void init_LCM(void)
```

```
{ write_inst(0x20);                    //设定功能
  write_inst(0x00);                    //设定功能
  write_inst(0x20);                    //设定功能
  write_inst(0x00);                    //设定功能
  write_inst(0x20);                    //设定功能
  write_inst(0x00);                    //设定功能
  write_inst(0x20);                    //设定功能
  write_inst(0x80);                    //设定功能
  write_inst(0x00);                    //关闭显示功能
  write_inst(0x08);                    //关闭显示功能
  write_inst(0x00);                    //清除显示屏
  write_inst(0x01);                    //清除显示屏
  write_inst(0x00);                    //设定输入模式
  write_inst(0x06);                    //设定输入模式
  write_inst(0x00);                    //开启显示功能
  write_inst(0x0e);                    //开启显示功能
}                                      //函数结束
```

检查忙碌函数

沟通是双方面的事，当8051对LCM下指令或丢数据时，LCM不见得有空处理，所以在前面的write_inst函数或write_data函数的最后，都会利用“for （i=0;i<10;i++）；”指令以缓冲数据处理或传输。其实我们可敲敲LCM的门，问它忙不忙？只要读取DB7，也就是BF标志位，若BF=1表示LCM还没处理完成，不要再继续丢数据或下指令给它。因此，在此就以一个简单的检查忙碌函数，以确认可继续下一步的操作，如下所示。

```
//====检查忙碌函数==================================
void check_BF(void)
{ char x = 0x80;                       //声明变量
  while(x&0x80)                        //若BF=1则执行循环
  {     RS = 0; RW=1;                  //读取指令模式
        en = 1;                        //启用
        x = P1;                        //读取指令
        en = 0;                        //完成
  }                                    //循环结束
}                                      //函数结束
```

写入数据函数

8051对LCM的沟通，除了写入指令外，写入数据也是很常用的操作，将这项操作写

成一个函数，就能使程序更简洁，如下所示。

```
//====写入数据函数==================================
void write_char(char character)
{ RS = 1; RW=0;                          // 写入数据模式
  en = 1;                                // 启用
  P1 = character;                        // 写入字符
  en = 0; check_BF();                    // 完成
}                                        // 函数结束
```

写入指定位置函数

对2列每列16个字的LCM而言，我们可将写入指令定义得更完善，让所要显示的字符能“对号入座”。如下所示，在函数的自变量里，增加line与location两个自变量，line就是第几列，0代表第一列、1代表第二列；location则为第几个位置，有效的位置为0~15，整个函数如下。

```
//====写入指定位置函数==============================
void display_char(char line,location,character)
{ RS = 0; RW= 0;                         // 写入指令模式
  en = 1;                                // 启用
  if(line=0)
       P1 = 0x80+location;               // 写入第一列
  If(line=1)
       P1 = 0x90+location;               // 写入第二列
  en = 0; check_BF();                    // 完成
  write_char(character);                 // 写入字符
}                                        // 函数结束
```

整列空白函数

若要将整列填入空白，可使用下列函数，其中的line自变量为所要填入空白的列，0代表第一列，1代表第二列，整个函数如下。

```
//==== 整列空白函数 ============================
void blank_line(char line)
{ RS = 0; RW= 0;                         // 写入指令模式
  en = 1;                                // 启用
  if(line=0)
       P1 = 0x80;                        // 写入第一列
```

```
If(line=1)
    P1 = 0x90;                          // 写入第二列
en = 0; check_BF();                     // 完成
for(char i=0;i<16;i++)
    write_char( ‘‘);                    // 写入字符
}                                       // 函数结束
```

13.5 LCM与8051的连接

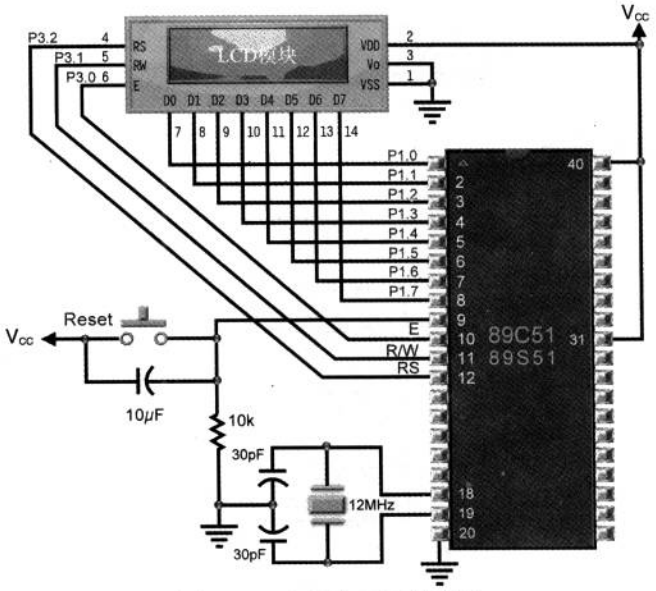


图13-11 LCM与8051的连接

如图13-11所示，LCM的数据总线D0-D7可与8051的Port 1，Port 2或Port 3连接。若要与8051的Port 0连接，其中每一条线必须各连接一个上拉电阻，比较麻烦！而LCM的三条控制线E、R/W与RS则可连接到其他没被用到的输入/输出口，同样地，为避免麻烦，最好不要把三条控制线连接到Port 0。

至于LCM的电源引脚VCC、接地引脚VSS以及明亮度控制引脚Vo，最简单的连接方法是将电源引脚连接+5V，接地引脚与明亮度控制引脚接地即可。

13.6 实例演练

在本单元里提供两个实例，如下所示。

13.6.1 LCD文字显示实例演练

实验要点

如图13-11所示，在本单元里将进行简单的LCD文字显示的实验。效果是首先在第一列里显示“LCM test program”，2s后在第二列里显示“Everything is OK”；再经2s后在第一列里改为“中文LCM 测试程序”，第二列里改为“一切正常欢迎使用”…，如图13-12所示。

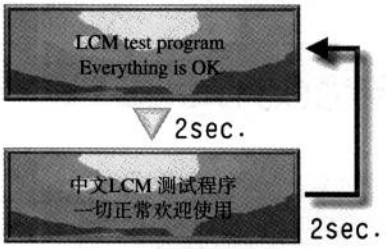
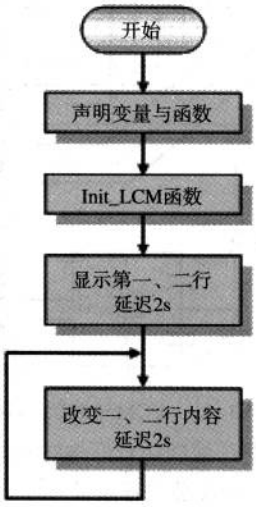


图13-12 功能示意图

流程图与程序设计




```
/*LCD文字显示实验(ch13-1.c)*/
#include <reg51.h>
#define LCDP   P1                    // LCD接至P1
sbit RS=P3^2;   sbit RW=P3^1;  sbit en=P3^0;
//sbit RS=P1^0; sbit RW=P1^1;  sbit en=P1^2;
char line1[]="LCM test program";
char line2[]="Everything is OK";
char line3[]="中文LCM 测试程序";
```

```
char line4[]="一切正常欢迎使用";
void init_LCM(void);
void write_inst(char);
void write_char(char);
void check_BF(void);
void delay1ms(int);
main()
{ char i;                                  // 声明变量
  init_LCM();                              // 初始设定
  while(1)                                 // 无尽循环
//=====LCM test program ======
  {     write_inst(0x80);                  // 指定第一列位置
        for(i=0;i<16;i++)                  // 循环
          write_char(line1[i]);            // 显示16个字
//=====Everything is OK ======
        write_inst(0x90);                  // 指定第二列位置
        for(i=0;i<16;i++)                  // 循环
          write_char(line2[i]);            // 显示16个字
        delay1ms(2000);                    // 延迟2s
//=====中文LCM 测试程序 ======
        write_inst(0x80);                  // 指定第一列位置
        for(i=0;i<16;i++)                  // 循环
          write_char(line3[i]);            // 显示16个字
//=====一切正常欢迎使用 ======
        write_inst(0x90);                  // 指定第二列位置
        for(i=0;i<16;i++)                  // 循环
          write_char(line4[i]);            // 显示16个字
        delay1ms(2000);                    // 延迟2s
}                                          // while结束
}                                          // 程序结束
//====初始设定函数(8位传输模式)==================
void init_LCM(void)
{ write_inst(0x30);                        // 设定功能
  write_inst(0x30);                        // 设定功能
  write_inst(0x30);                        // 设定功能
  write_inst(0x38);                        // 设定两列
  write_inst(0x08);                        // 关闭显示功能
  write_inst(0x01);                        // 清除显示屏
```

```
  write_inst(0x06);                    // 设定输入模式
  write_inst(0x0e);                    // 开启显示功能
}                                      // 函数结束
//====写入指令函数==================================
void write_inst(char inst)
{ RS = 0; RW=0;                        // 写入指令模式
  en = 1;                              // 启用
  LCDP = inst;                         // 写入指令
  en = 0; check_BF();                  // 完成
}                                      // 函数结束
//====写入数据函数==================================
void write_char(char character)
{ RS = 1; RW=0;                        // 写入数据模式
  en = 1;                              // 启用
  LCDP = character;                    // 写入字符
  en = 0;check_BF();                   // 完成
}                                      // 函数结束
//====检查忙碌函数==================================
void check_BF(void)
{
  char i,x = 0x80;                     // 声明变量
  while(x&0x80)                        // 若BF=1则执行循环
  {     RS = 0; RW=1;                  // 读取指令模式
        en = 1;                        // 启用
        x = LCDP;                      // 读取指令
        en = 0;
        for(i=0;i<10;i++);             // 完成
  }                                    // 循环结束
}                                      // 函数结束
//====延迟函数======================================
void delay1ms(int x)
{ int i,j;                             // 声明变量
  for(i=1;i<x;i++)                     // 外循环
        for(j=1;j<120;j++);            // 内循环
}                                      // 函数结束
```

LCD文字显示实验（ch13-1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查原始程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图13-11连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

13.6.2 自编字型图案实例演练

实验要点

如图13-11所示为本实验所要采用的电路图，在此我们将自建两个字型，分别是“▲”及“▼”，其中的“▲”代表上午、“▼”代表下午，在LCM里，第一列将显示以“时:分:秒”的格式显示时间，每秒钟改变一次显示，在其右边将以自建字型“▲”、“▼”区别上下午，如图13-13所示。



图13-13 功能示意图

流程图与程序设计

根据功能需求，在此分成三部分来说明，第一部分是自建字型，第二部分是时间的产生，第三部分是将计时数转换成显示数据。

- 在自建字型方面，如图13-14所示为本次所要建的字型，其中的“▲”编码为“0x04、0x0e、0x1f、0x00、0x00、0x00、0x00、0x00”；而“▼”编码为“0x00、0x00、0x00、0x00、0x00、0x1f、0x0e、0x04”，在此将它们存入pat[16]数组，如下所示。

```
char pat[16]={  0x04,0x0e,0x1f,0x00,0x00,0x00,0x00,0x00,
                0x00,0x00,0x00,0x00,0x00,0x1f,0x0e,0x04};
```

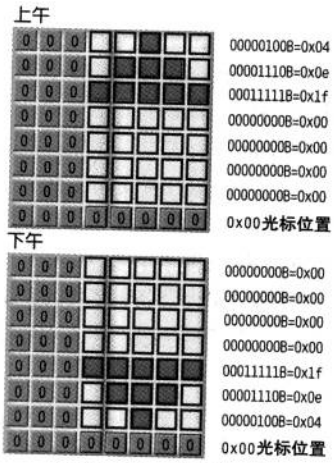


图13-14 自建字型

CG RAM的起始地址为01000000B，即0x40，每8个地址为一个自建字型。在程序里，可以使用下列指令将这个数组的内容填入CG RAM。

```
RS = 0; RW= 0;                    // 写入指令模式
en = 1;                           // 启用
P1 = 0x40;                        // 指定CGRAM的第一个位置
en = 0; check_BF();               // 完成
RS = 1; RW=0;                     // 写入数据模式
for(char i=0;i<16;i++)
  write_char(char pat[i]);        // 写入字符
```

- 时间的产生是利用TIMER0中断，每次中断为50ms，每20次中断就是1秒钟，因此要调整时间的输出。在此，hour变数为“时”数、minute变数为“分”数、second变数为“秒”数。若秒数超过60秒，则秒数归零，进而调整分数（即分+1）；若分数超过60分，则分数归零，进而调整时数（实时+1）；若时数超过12时，则时数归零，进而改变上下午状态（即0变1、1变0），如下。

```
void clock(void) interrupt 1      // T0中断子程序
{ TH0=(56636-50000)/256;          // 填入计时量
  TL0=(56636-50000)%256;          // 填入计时量
  If(--count=0)                   // 中断次数是否达到20次
  {     count=20;                 // 重新计次
        second++;                 // 秒数加1
        if(second=>60)            // 是否达到60秒
        { second=0;               // 秒数归零
```

```
        minute++;                    // 分数加1
        if(minute=>60)               // 是否达到60分
        {   minute=0;                // 分数归零
            hour++;                  // 时数加1
            if(hour=>13)             // 是否达到13小时
            { hour=0;                // 时数归零
              ampm=~ampm;}           // 切换上下午
  }   } }                            // 结束
```

- 紧接着是将hour、minute、sceond、ampm转换成在LCM显示的ASCII码及自建的图案。hour、minute、sceond变量是以16进位数字储存时数、分数与秒数，我们可利用“/10”萃取其十位数，利用“%10”萃取其个位数；再加上0x30，则数字就变成ASCII，然后把它存入显示区数组d[]，如图13-15所示。

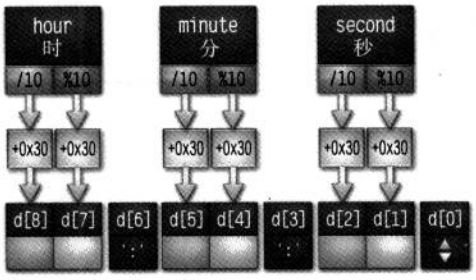
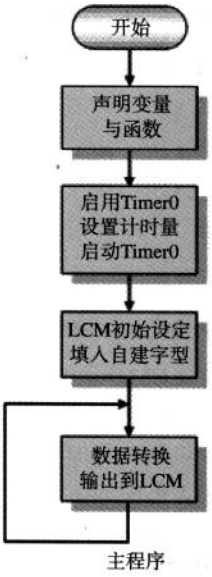
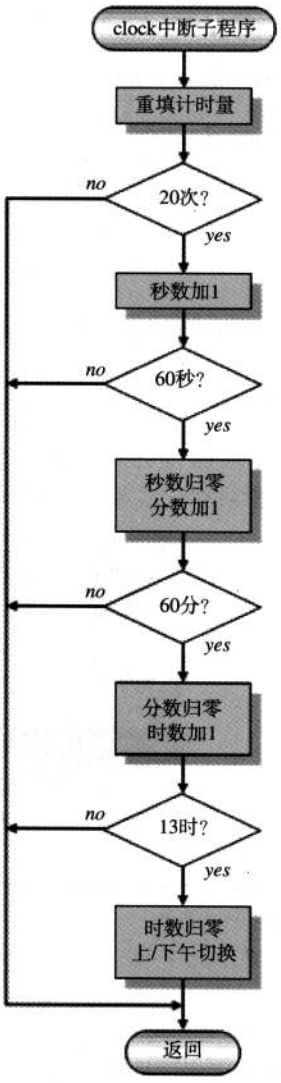


图13-15 数字的时分秒转换成ASCII码

```
d[8]=hour/10+0x30;                   // 时数的十位数显示数据
d[7]=hour%10+0x30;                   // 时数的个位数显示数据
d[6]=':';                          // 显示冒号
d[5]=minute/10+0x30;                 // 分数的十位数显示数据
d[4]=minute%10+0x30;                 // 分数的个位数显示数据
d[3]=':';                          // 显示冒号
d[2]=second/10+0x30;                 // 秒数的十位数显示数据
d[1]=second%10+0x30;                 // 秒数的个位数显示数据
if(ampm==0)  d[0]=0;                 // 上午
else d[0]=1;                         // 下午
```

流程图与整个程序如下所示。






```
/*自编字型图案实验(ch13-2.c)*/
#include  <reg51.h>
#define LCDP    P1                    // LCD接至P1
sbit RS=P3^2;   sbit RW=P3^1;  sbit en=P3^0;
char      count=20;
char  d[]={'0','0',':','0','0',':','0','0',0 };       // 声明显示区数组变量
/* 声明自建字型数组变量 */
char pat[16]={   0x04,0x0e,0x1f,0x00,0x00,0x00,0x00,0x00,
                0x00,0x00,0x00,0x00,0x00,0x1f,0x0e,0x04};
```

```
bit ampm=0;                                  // 声明变量
char hour=0;                                 // 声明变量
char minute=0;                               // 声明变量
char second=0;                               // 声明变量
void transfer(void);                         // 声明转换函数
void write_inst(char);                       // 声明写入指令函数
void write_char(char);                       // 声明写入字符函数
void write_pat(void);                        // 声明写入自建字型函数
void check_BF(void);                         // 声明检查忙碌函数
void init_LCM(void);                         // 声明LCM初始设定函数
//=======================================
main()
{ char i;
  EA=1; ET0=1;                               // 启用Timer 0中断
  TH0=(56636-50000)/256;                     // 填入计时量
  TL0=(56636-50000)%256;                     // 填入计时量
  TR0=1;                                     // 启动Timer 0
  init_LCM();                                // 初始设定
  write_pat();                               // 写入自建字型
  while(1)                                   // 无穷循环
  {     transfer();                          // 数据转换
        for(i=0;i<9;i++);                    // 显示
         {      write_inst(0x80+i);          // 指定位置
                write_char(d[8-i]);}         // 写入字符
  }                                          // while结束
}                                            // 结束
//====转换函数==================
void transfer(void)
{ d[8]=hour/10+0x30;                         // 时数的十位数显示数据
  d[7]=hour%10+0x30;                         // 时数的个位数显示数据
  d[6]=':';                                  // 显示冒号
  d[5]=minute/10+0x30;                       // 分数的十位数显示数据
  d[4]=minute%10+0x30;                       // 分数的个位数显示数据
  d[3]=':';                                  // 显示冒号
  d[2]=second/10+0x30;                       // 秒数的十位数显示数据
  d[1]=second%10+0x30;                       // 秒数的个位数显示数据
  if(ampm==0)  d[0]=0;                       // 上午
  else d[0]=1;                               // 下午
```

```
}                                          // 函数结束
//====写入指令函数==================================
void write_inst(char inst)
{ RS = 0; RW=0;                            // 写入指令模式
  en = 1;                                  // 启用
  LCDP = inst;                             // 写入指令
  en = 0; check_BF();                      // 完成
}                                          // 函数结束
//====写入数据函数==================================
void write_char(char character)
{ RS = 1; RW=0;                            // 写入数据模式
  en = 1;                                  // 启用
  LCDP = character;                        // 写入字符
  en = 0;check_BF();                       // 完成
}                                          // 函数结束
//====写入自建字型函数==============================
void write_pat(void)
{ char i;
  RS = 0; RW= 0;                           // 写入指令模式
  en = 1;                                  // 启用
  LCDP = 0x40;                             // 指定CGRAM的第一个位置
  en = 0; check_BF();                      // 完成
  RS = 1; RW=0;                            // 写入数据模式
  for(i=0;i<16;i++)
        write_char(pat[i]);                // 写入字符
}                                          // 函数结束
//====检查忙碌函数==================================
void check_BF(void)
{ char i,x = 0x80;                         // 声明变量
  while(x&0x80)                            // 若BF=1则执行循环
  {     RS = 0; RW=1;                      // 读取指令模式
        en = 1;                            // 启用
        x = LCDP;                          // 读取指令
        en = 0; for(i=0;i<10;i++);         // 完成
  }                                        // 循环结束
}                                          // 函数结束
//====初始设定函数(8位传输模式)====================
void init_LCM(void)
```

```
{ write_inst(0x30);                    // 设定功能
  write_inst(0x30);                    // 设定功能
  write_inst(0x30);                    // 设定功能
  write_inst(0x38);                    // 设定两列
  write_inst(0x08);                    // 关闭显示功能
  write_inst(0x01);                    // 清除显示屏
  write_inst(0x06);                    // 设定输入模式
  write_inst(0x0e);                    // 开启显示功能
}                                      // 函数结束
//===== Timer 0中断子程序 ========================
void clock(void) interrupt 1           // T0中断子程序
{ TH0=(56636-50000)/256;               // 填入计时量
  TL0=(56636-50000)%256;               // 填入计时量
  while(--count==0)                    // 中断次数是否达到20次
  {     count=20;                      // 重新计次
        second++;                      // 秒数加1
        if(second>=60)                 // 是否达到60秒
        {second=0;                     // 秒数归零
         minute++;                     // 分数加1
         if(minute>=60)                // 是否达到60分
         {      minute=0;              // 分数归零
                hour++;                // 时数加1
                if(hour>=13)           // 是否达到13小时
                { hour=0;              // 时数归零
                  ampm=~ampm;}         // 切换上下午
} }    } }                             // 结束
```

自编字型图案实验（ch13-2.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查原始程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图13-11连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实

体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

如何结合本实验里的LCM及第11章所介绍的ADC制作一个LCD显示的数字温度表，电路图可参考图13-16。

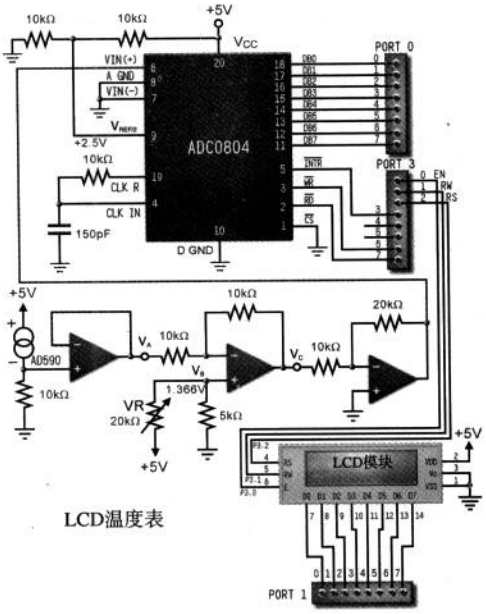
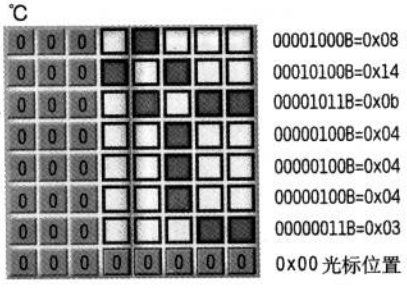


图13-16　LCD数字温度表

在LCD显示屏中，请自制“℃”字型，如图13-16所示。



13.7 实时练习



在本章中探讨了LCM的结构、指令与应用方法。在此请试着回答下列问题，以确认

对于此部分的掌握程度。

选择题

（ ）1. 若要LCM显示某些字符，则需要把所要显示的字符放入何处？
（A）CG RAM （B）DDRAM （C）IRAM （D）GDRAM

（ ）2. 若要读取LCM的状态，则应如何设定？
（A）RS=0、R/$\overline{W}$=0 （B）RS=1、R/$\overline{W}$=0
（C）RS=1、R/$\overline{W}$=1 （D）RS=0、R/$\overline{W}$=1

（ ）3. 若要对LCM下指令，则应如何设定？
（A）RS=0、R/$\overline{W}$=0 （B）RS=1、R/$\overline{W}$=0
（C）RS=1、R/$\overline{W}$=1 （D）RS=0、R/$\overline{W}$=1

（ ）4. 若要将数据写入LCM，则应如何设定？
（A）RS=0、R/$\overline{W}$=0 （B）RS=1、R/$\overline{W}$=0
（C）RS=1、R/$\overline{W}$=1 （D）RS=0、R/$\overline{W}$=1

（ ）5. 若要检查LCM是否忙碌，则应如何设定？
（A）RS=0、R/$\overline{W}$=0 （B）RS=1、R/$\overline{W}$=0
（C）RS=1、R/$\overline{W}$=1 （D）RS=0、R/$\overline{W}$=1

（ ）6. 若LCM更明亮，应如何处理？
（A）Vo引脚调往高电压 （B）Vo引脚调往低电压
（C）加大电源电压 （D）降低电源电压

（ ）7. 若对LCM操作，对应EN引脚做何操作？
（A）送入一个正脉冲 （B）送入一个负脉冲
（C）EN引脚接地即可 （D）EN引脚不影响

（ ）8. 中文LCM的中文字型放置在哪里？
（A）CG ROM （B）HCG ROM （C）DDRAM （D）GD RAM

（ ）9. 中文LCM-WG12864J-YYH-N的面板为？
（A）彩色LCD面板 （B）144 × 32 LCD面板
（C）128 × 64 LCD面板 （D）144 × 64 LCD面板

（ ）10. 中文LCM-WG12864J-YYH-N采用哪个控制器？
（A）HD44780 （B）ST7920 （C）WG12864 （D）以上皆非

问答题

1. 常用的以HD44780控制器所组成的LCM，有哪几种显示模式？
2. 请写出常用LCM的引脚，其中用于调整明亮度的是哪一只引脚？
3. 常用LCM共有14只引脚，有哪两种封装方式？
4. 试述LCM初始化的步骤。
5. 若要对LCM下指令，必须等它有空，如何侦测LCM是否有空？

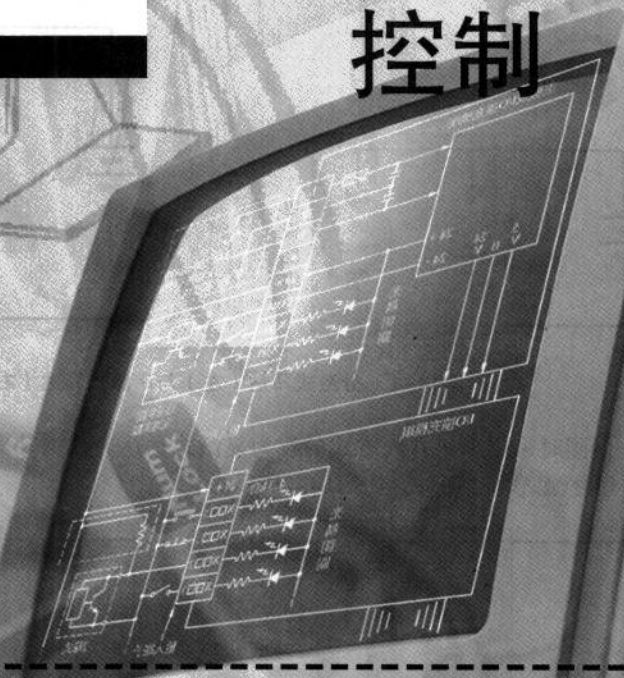
6. 常用的LCM拥有多少CG RAM？多少CG ROM？可自建多少个字型？

7. 当我们建好字型后，如何送入LCM？

8. 常用的中文LCM-WG12864J-YYH-N，使用哪个控制器？

9. 试简述中文LCM里，DD RAM、CG RAM、CG ROM、HCG ROM和IRAM和GD RAM的功能。

第14章 直流马达的控制



本章内容丰富，主要包括以下两部分。

硬件部分：

认识直流马达的结构。

认识直流马达的驱动方式。

程序与实作部分：

直流马达的方向控制，直流马达的PWM驱动程序。

14.1 认识直流马达



直流马达的结构可分为机壳（enclosure）、定子（stator）与转子（即电枢，armature）。大中型的直流马达的定子与转子上各有绕组（线圈），这两种绕组之间，可采用串联或并联方式。如图14-1所示，采用串联方式的直流马达称为串激式马达；采用并联方式的马达称为分激式马达。

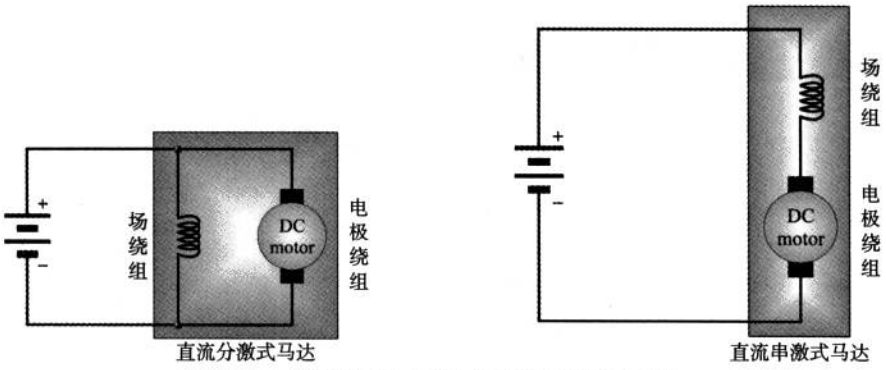


图14-1 分激式马达（左）与串激式马达（右）

另外，同时采用串、并联方式的称为复激式马达，根据其串、并方式又分为长复激式马达与短复激式马达两种，如图14-2所示。

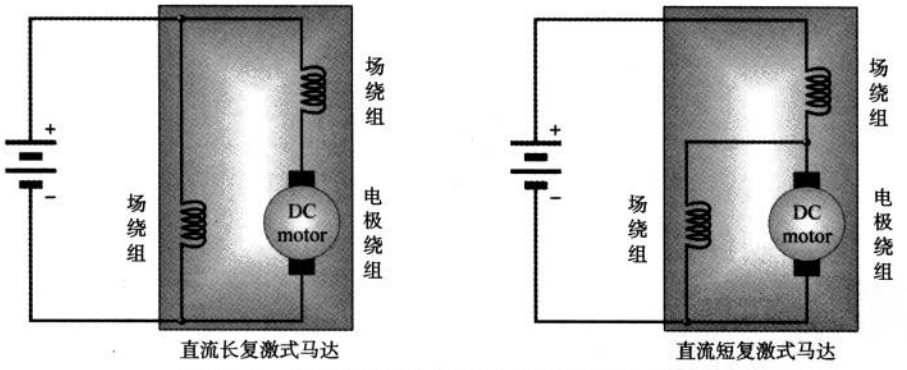


图14-2 长复激式马达（左）与短复激式马达（右）

以分激式马达为例，定子绕组的激磁方向与转子绕组的激磁方向决定了转动的方向，若单独将定子绕组的激磁方向改变，或单独将转子绕组的激磁方向改变，则其旋转方向将与原来的旋转方向相反。

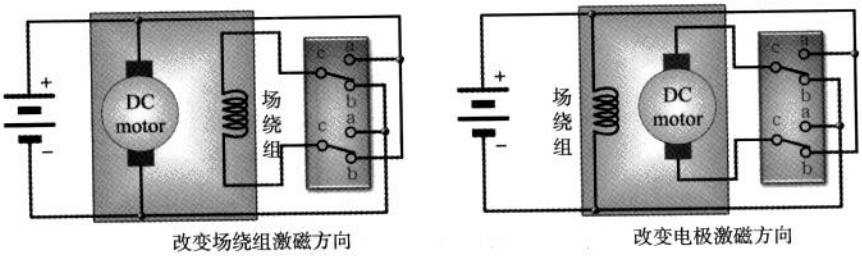


图14-3 改变分激式马达的旋转方向

对于小型的直流马达而言，其定子部分采用永久磁铁，而不使用绕组。换言之，其中只有一个绕组，也就是转子绕组，其定子的磁场方向是固定的。如此一来，若要改变马达的转向，只要改变其外加电源的方向即可。

14.2 直流马达的驱动方式

直流马达的驱动方式，就是把直流电源加到直流马达上，使之旋转。下面将以永久磁铁为定子磁场的中小型直流马达为例进行介绍。

用继电器驱动直流马达

如图14-4所示，微控制器信号连接到晶体管，以控制继电器。当微控制器送出一个高电平信号，即可产生i_b、i_c，继电器激磁，而继电器的a-c接点将接通，即可提供直流马达电源，使之旋转。其中的V_{cc}不一定是5V电源，而是根据继电器及直流马达的规格，取用适当的电压。一般来说，电功率$P= V \times G$，v越大功率越大；即便是相同的功率，v越大i越小，损失越小。

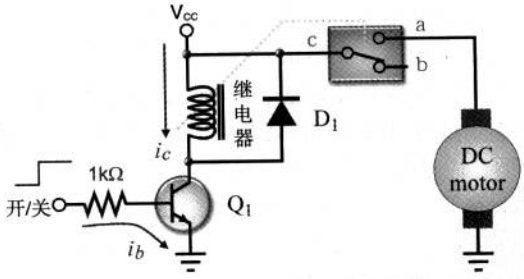


图14-4 用继电器开关直流马达

以晶体管驱动直流马达

如图14-5所示，微控制器信号连接到达林顿晶体管（Darlington transistor），直接提供直流马达的电源，使之旋转。其中的D_1、D_2二极管的功能是为了保护达林顿晶体管，VCC也不一定是5V电源，可根据直流马达的规格，取用较高的电压。而此电路不但可以控制直流马达的开或关，还可以控制其功率大小，以达到转速控制的目的。

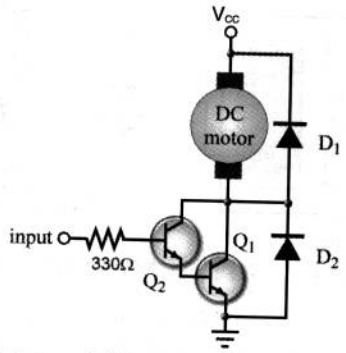


图14-5　以达林顿晶体管驱动直流马达

以继电器控制直流马达的方向

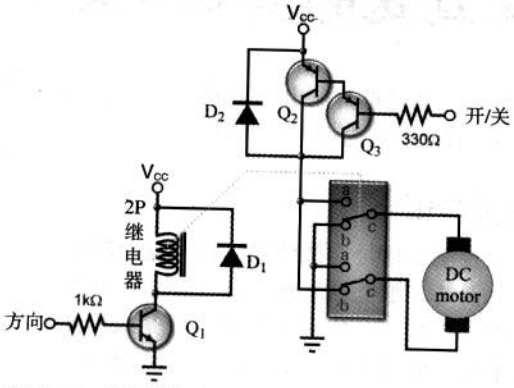


图14-6　以达林顿晶体管与继电器控制直流马达

如图14-6所示，微控制器信号连接到达林顿晶体管与继电器，其中的继电器是2P继电器，同时提供两组c接点，由微控制器信号连接到“方向”的引脚，即可驱动Q_1晶体管，以控制继电器。当“方向”引脚上有高电平信号时，继电器激磁，两组c-a接点接通，而直流马达上方连接到Q_2、Q_3所组成的达林顿晶体管，所以此时直流马达上方连接到正电源，另外，直流马达下方通过另一组c-a接点接地。

若“方向”引脚上有低电平信号时，继电器消磁，两组c-b接点接通，直流马达上方通过c-b接点接地；而直流马达下方通过另一组c-b接点连接到Q_2、Q_3所组成的达林顿晶体管，所以此时直流马达下方连接到正电源。

若直流马达上方接电源，下方接地，将使其顺时针旋转，此时，如果颠倒其接线，直流马达上方接地、下方接电源，将使其逆时针旋转。

另外，我们也可通过“开/关”引脚来决定该马达是否旋转。即我们不但可控制直流马达的开与关，也可控制其转向。

以晶体管控制直流马达的方向

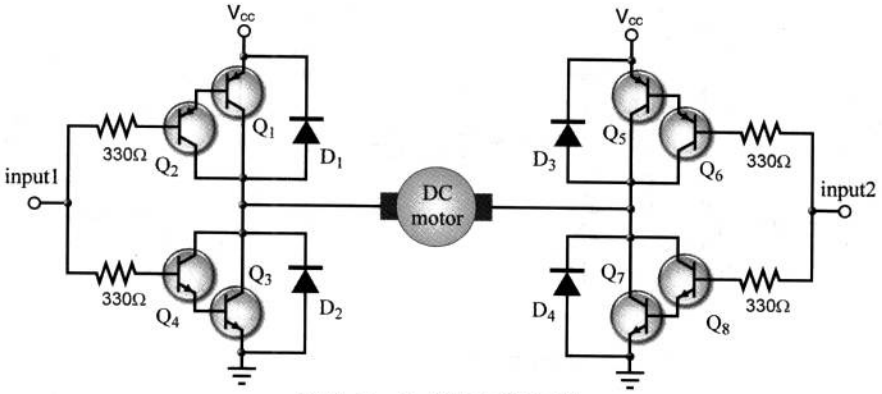


图14-7 桥式驱动直流马达

如图14-7所示，Q_1、Q_2是一组PNP型达林顿晶体管，Q_3、Q_4是一组NPN型达林顿晶体管，Q_5、Q_6是一组PNP型达林顿晶体管，Q_7、Q_8是一组NPN型达林顿晶体管，不管是NPN型达林顿晶体管还是PNP型达林顿晶体管，都可找到现成、配对的商品，而且不贵！若使用现成达林顿晶体管，电路就非常简单，而且可靠！电路的左、右对称，动作也类似。不管是左边电路还是右边电路，当微控制器送一个高电平信号到input1或input2端时，上方的PNP达林顿晶体管截止，而上方的NPN达林顿晶体管导通；当微控制器送一个低电平信号到input1端时，上方的PNP达林顿晶体管导通，而上方的NPN达林顿晶体管截止。

若送一个高电平信号到input1端，同时送一个低电平信号到input2端时，则电流由右而左流过此直流马达，如图14-8所示。

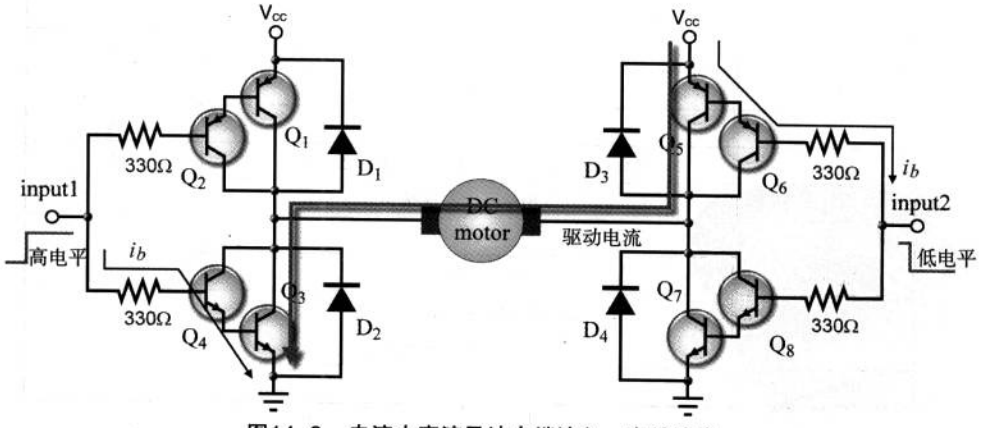


图14-8 电流由直流马达右端流入、左端流出

反之，若送一个低电平信号到input1端，同时送一个高电平信号到input2端时，则电流由左而右流过此直流马达，如图14-9所示。

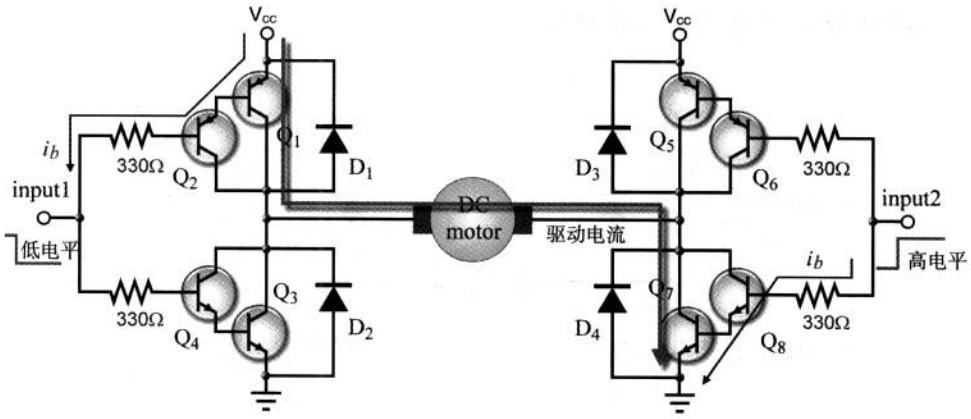


图14-9　电流由直流马达左端流入、右端流出

若直流马达上方接电源，下方接地，将使其顺时针旋转；此时，如果颠倒其接线，直流马达上方接地，下方接电源，将使其逆时针旋转。

● 互补（配对）达林顿功率晶体管－TIP12x系列

互补达林顿功率晶体管（Darlington Complementary Silicon Power Transistors）是一种实用且便宜的中型功率晶体，首先要介绍的是TIP12*x*系列，这一系列包括三组配对，分别是TIP-120（NPN）与TIP-125（PNP）、TIP-121（NPN）与TIP-126（PNP）、TIP-122（NPN）与TIP-127（PNP），其常见规格如表14-1所示。

表14-1

特性	TIP120 TIP125	TIP121 TIP126	TIP122 TIP127	单位
V_{CEO}	60	80	100	V
V_{CBO}	60	80	100	V
V_{EBO}	5.0			V
I_C	5.0			A
I_{CP}	8.0（脉冲，300μs、Duty cycle≤2.0%）			A
I_B	120（最大、连续）			mA
P_D	65			W
h_{FE}(DC)	1000（最小）			-
$V_{CE(sat)}$	2.0（I_C=3.0A、I_B=12mA） 4.0（I_C=5.0A、I_B=20mA）			V
$V_{BE(on)}$	2.5			V

TIP12*x*系列的内部电路结构如图14-10所示。

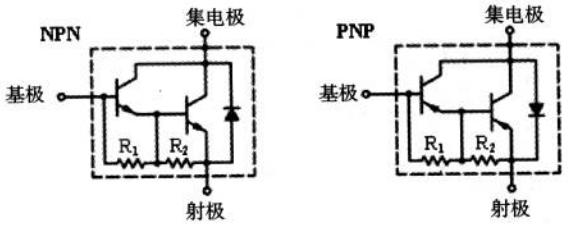


图14-10 TIP12x系列内部电路结构

其中R_1约为10kΩ，R_2约为150Ω。而其包装采用扁平的TO-220包装，其外观和引脚配置与尺寸如图14-11和图14-12所示。

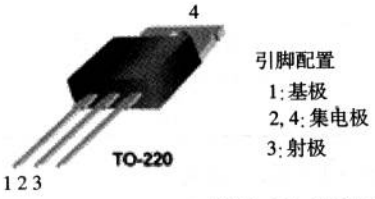


图14-11 TIP12x系列的外观与引脚配置

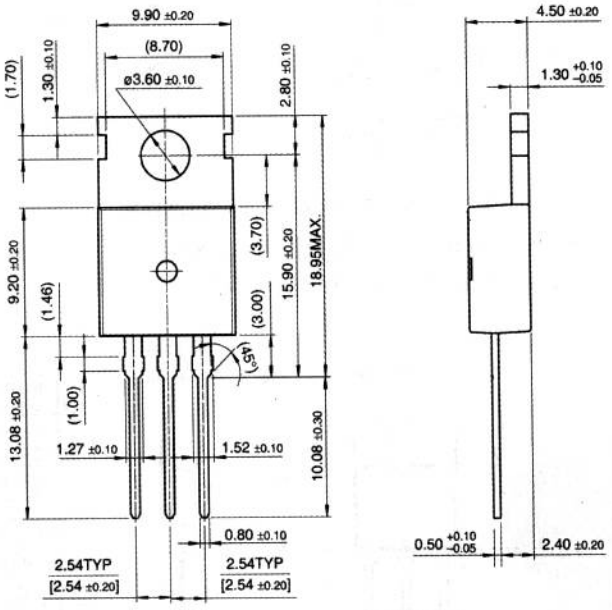


图14-12 TIP12x系列的尺寸

● 互补（配对）达林顿功率晶体管－TIP14x系列

前述TIP12*x*系列可提供5A电流，若还不够，则可采用TIP14*x*系列，这一系列包括三组配对，分别是TIP-140（NPN）与TIP-145（PNP）、TIP-141（NPN）与TIP-146（PNP）、TIP-142（NPN）与TIP-147（PNP），其常见规格如表14-2所示。

表14-2

特 性	TIP140 TIP145	TIP141 TIP146	TIP142 TIP147	单 位
V_{CEO}	60	80	100	V
V_{CBO}	60	80	100	V
V_{EBO}	5.0			V
I_C	10			A
I_{CP}	15（脉冲，5ms、Duty cycle≤10%）			A
I_B	0.5（最大、连续）			A
P_D	125			W
h_{FE}(DC)	1000（最小）			-
$V_{CE(sat)}$	2.0（I_C=5.0A、I_B=10mA） 3.0（I_C=10.0A、I_B=40mA）			V
$V_{BE(on)}$	3.0			V

TIP14x系列的内部电路结构如图14-13所示。

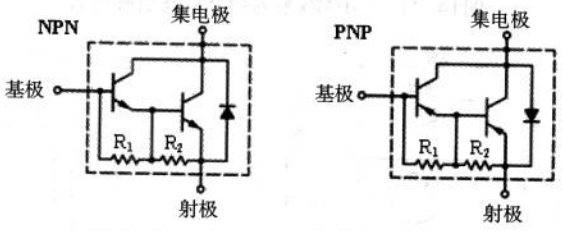
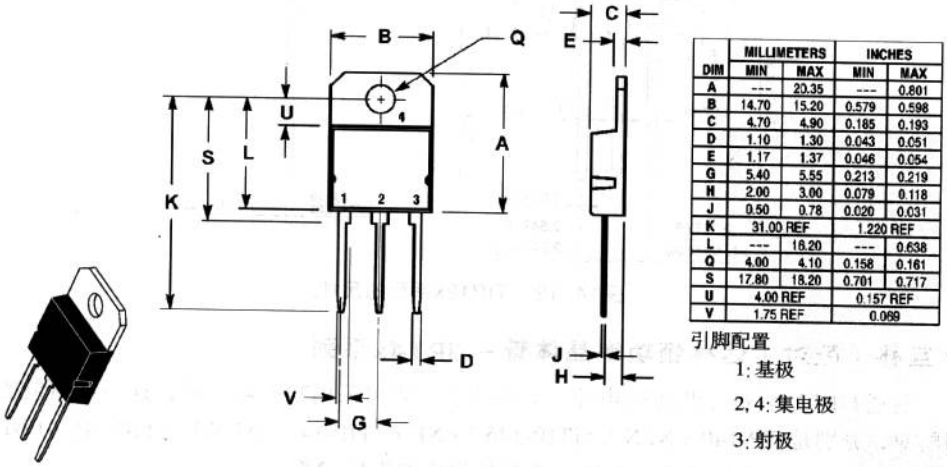


图14-13 TIP14x系列内部电路结构

其中R_1约为8kΩ，R_2约为40Ω。而其包装采用扁平的TO-218包装，也提供SOT-93的表面贴式包装，其外观、引脚配置与尺寸如图14-14所示。



DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	---	20.35	---	0.801
B	14.70	15.20	0.579	0.598
C	4.70	4.90	0.185	0.193
D	1.10	1.30	0.043	0.051
E	1.17	1.37	0.046	0.054
G	5.40	5.55	0.213	0.219
N	2.00	3.00	0.079	0.118
J	0.50	0.78	0.020	0.031
K	31.00 REF		1.220 REF	
L	---	16.20	---	0.638
Q	4.00	4.10	0.158	0.161
S	17.80	18.20	0.701	0.717
U	4.00 REF		0.157 REF	
V	1.75 REF		0.069	

引脚配置

1: 基极

2, 4: 集电极

3: 射极

图14-14 TIP14x系列的外观、引脚配置与尺寸

桥式达林顿功率晶体管模块－TA7257P系列

前述TIP12*x*与TIP14*x*系列可提供较大电流，且价格便宜。不过，应用在桥式电路上，需要4个（两对）晶体管，电路稍微复杂一点。若所驱动的直流马达不大，则可采用桥式达林顿功率晶体管模块，这是一种将两对达林顿晶体管包装在一起，并内含控制电路与保护电路的装置，让应用电路简化。而市面上这种模块很多，以比较常见的日系Toshiba厂牌的TA7257P为例进行介绍，其内部结构图如图14-15所示。

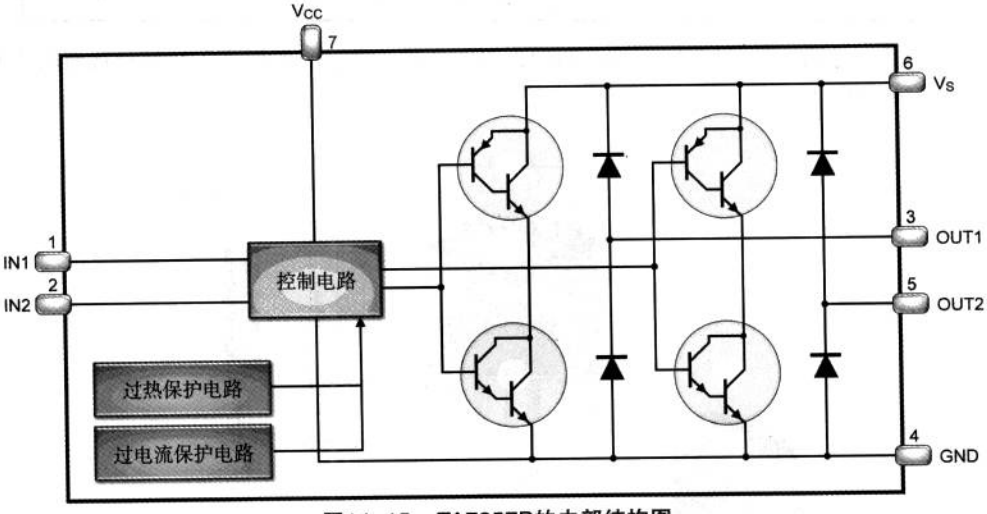


图14-15 TA7257P的内部结构图

其基本规格与功能如下。

- 输出电流的平均值可达1.5A（连续），而峰值电流更可达4.5A。
- 输入电压：V_{CC}=6~18V、V_S=0~18V，其中V_{CC}提供控制电路的电源、V_S提供负载的电源。
- 提供4种操作模式：正转（CW）、反转（CCW）、停止与刹车。
- 内含过热保护电路与过电流保护电路。
- TA7257P的引脚如表14-3所示。

表14-3

引脚号码	引脚名称	说 明
1	IN1	输入引脚1
2	IN2	输入引脚2
3	OUT1	输出引脚1
4	GND	接地引脚
5	OUT2	输出引脚2
6	V_s	负载电源引脚
7	V_{cc}	控制电路电源引脚

• TA7257P的功能如表14-4所示。

表14-4

IN1引脚	IN2引脚	OUT1引脚	OUT2引脚	说 明
1	1	低电平	低电平	刹车
0	1	低电平	高电平	正转（反转）
1	0	高电平	低电平	反转（正转）
0	0	高阻抗	高阻抗	停止

当然，这么方便的零件，比4个TIP12x系列还贵一些（百元以下）！其外观与尺寸如图14-16所示。

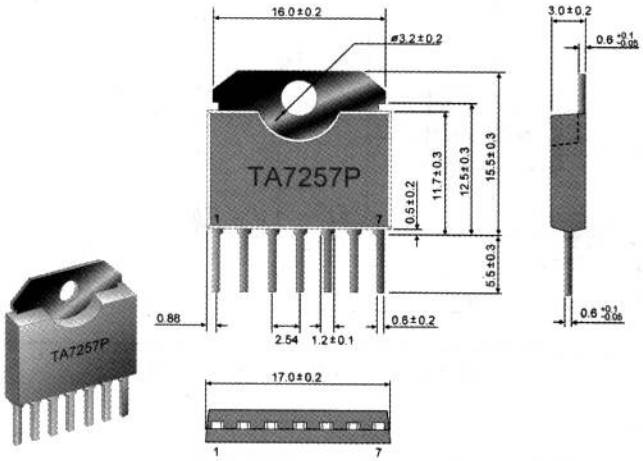


图14-16 TA7257P的外观图（左）与尺寸图（右）

当我们要应用TA7257P来驱动直流马达时，将1、2两脚连接到8051的Port，由这个Port传递控制信号；将3、5脚连接到所要驱动的直流马达，而在这两脚之间并接一个RC串联电路，其中的R=33Ω，C=0.1μF；第4脚接地，第6、7脚接V_{cc}，而靠近第7脚处并接一个10μF电容器到地，如图14-17所示。

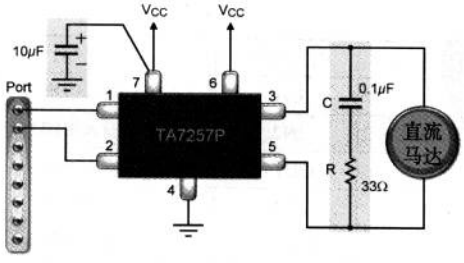


图14-17 TA7257P的应用电路

14.3 直流马达的PWM控制

驱动直流马达的电流大小将影响马达的输出转距与转速，而使用晶体管来控制直流马达的电流时，晶体管可能工作在动作区。如此一来，晶体管上的V_{CE}与电流I_C不小，所以晶体管上的功率损耗$P_D=V_{CE}\times I_C$当然就很可观！因此，采用这种线性的控制方式效率不高！

直流马达的功率采用平均值，当电压固定时，只要改变电流的平均值，即可改变输入功率，如图14-18所示，其中A脉冲的平均值为0.5A，相当于持续的直流电流0.5A；B脉冲的平均值为0.25A，相当于持续的直流电流0.25A。
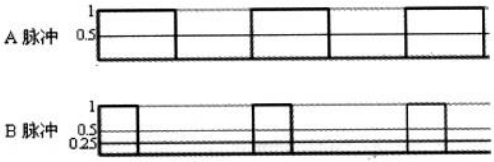


图14-18 脉冲的平均值

由上图可看出，在A脉冲之中，约有一半的时间是1A，一半的时间是0A，或者说约有一半的时间晶体管全开，一半的时间是晶体管全关，如此就能得到0.5A的平均值。在B脉冲之中，约有1/4的时间是晶体管全开，3/4时间是晶体管全关，如此就能得到0.25A的平均值。“晶体管全开”就是晶体管工作在饱和状态，“晶体管全关”就是晶体管工作在截止状态，在这种情况下，晶体管的损失最少、效率最高。

这种以改变脉冲宽度控制平均值的方法，称为脉冲宽度调变（Pulse Width Modulation，简称PWM；也有人称之为Pulse Duration Modulation，即PDM）。

在模拟电路中，我们可以使用比较器将正弦波与三角波调变为PWM波，如图14-19所示为PWM调变电路及PWM调变示意图。

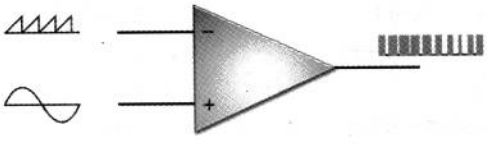


图14-19 PWM调变电路示意图

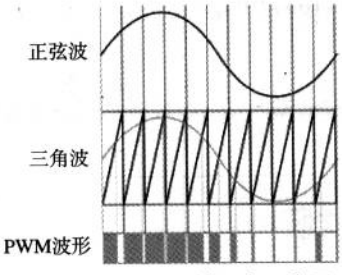


图14-20 PWM的调变示意图

若要把PWM波解调变成正弦波，则只要将PWM波接入积分电路即可变成正弦波。

在8051里，可以应用延迟函数来产生PWM波，以图14-5为例，若其input端连接到P1.0，我们可声明on变量，代表输出高电平的百分数，如下：

```
char on = 50;                          // 声明on的百分数(1到100)
sbit  input = P1^0;                    // 声明输出端口的位置
void delay1ms(unsigned char);          // 声明延迟函数
:
while(1)
{ input=1;                             // 输出高电平
  delay1ms(on);                        // 延迟on时间
  input=0;                             // 输出低电平
  delay1ms(100-on);                    // 延迟100-on时间
}                                      // 结束
```

如上所示，输出PWM波的工作周期（duty cycle）为50%，改变on的值，即可改变工作周期，如下所示的程序，从工作周期为10%开始，10s后，工作周期调整为20%，再过10s后，工作周期调整为30%，直到工作周期为100%。

```
char on;                               // 声明on的百分数(1到100)
sbit  input = P1^0;                    // 声明输出端口的位置
void delay1ms(unsigned char);          // 声明延迟函数
:
while(1)
{ on=10;                               // 指定初始值
  for(char k=0;k<10;k++)               // 重复10次循环
  {    for(char i=0;i<10;i++)          // for循环
       { input=1;                      // 输出高电平
         delay1ms(on+10*i);            // 延迟on时间
         input=0;                      // 输出低电平
         delay1ms(100- (on+10*i));     // 延迟100-on时间
  }      }                             // 结束for循环
}                                      // 结束
```

14.4 实例演练



在本单元里提供4个实例，如下所示。

14.4.1 继电器开关控制 实例演练

实验要点

在此将由8051的P1.0输出，通过晶体管以控制继电器；而由P2.0连接按钮开关PB0，当按下PB0时，继电器激磁，直流马达旋转；放开PB0时，继电器消磁，直流马达停止，如图14-21所示。

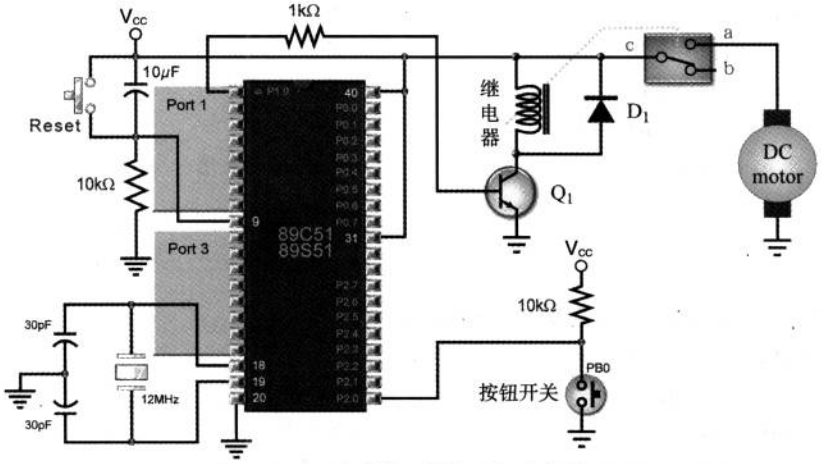
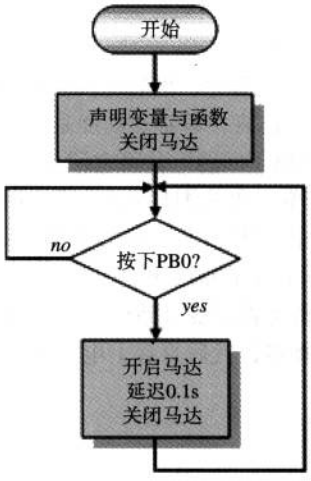


图14-21 继电器开关控制实验电路图

流程图与程序设计




```
/* 继电器开关控制实验(ch14-1.c) */
#include <reg51.h>
sbit motor = P1^0;                    // 声明直流马达位置
```

```
sbit PB0 = P2^0;                          // 声明按钮开关位置
void delay1ms(char);                      // 声明延迟函数
//=========================================
main()
{ motor=0;                                // 关闭直流马达
 PB0=1;                                   // 设定PB0
 while(1)                                 // 无穷循环
 {    if(PB0==0)                          // 若按下PB0
      { motor=1;                          // 开启直流马达
        delay1ms(100);                    // 旋转0.1s
        motor=0;                          // 关闭直流马达
      }
 }
}
// ========== 延迟函数 =========
void delay1ms(char x)
{       char i,j;
  for(i=0;i<x;i++)                        // 外循环
       for(j=0;j<120;j++);                // 内循环
}                                         // 延迟函数结束
```

继电器开关控制实验（ch14-1.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查原始程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图14-18连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

1. 在本实验里，若按住PB0不放，将会如何？

2. 在本实验里，有无抖动的困扰？若改变程序的延迟时间，让按一下PB0，直流马达旋转0.5s，感觉如何？

14.4.2 继电器的方向控制实例演练

实验要点

如图14-22所示，P1.1的功能是控制马达开/关的信号，若P1.1=0，马达可以转动；若P1.1=1则马达不动。P1.0的功能则是控制马达开的旋转方向，若P1.0=1，马达顺时针旋转；若P1.1=0，马达则逆时针旋转。按一下PB0钮，马达顺时针旋转0.5s；按一下PB1钮，马达逆时针旋转0.5s。若同时按住PB0与PB1钮，马达将不转动。

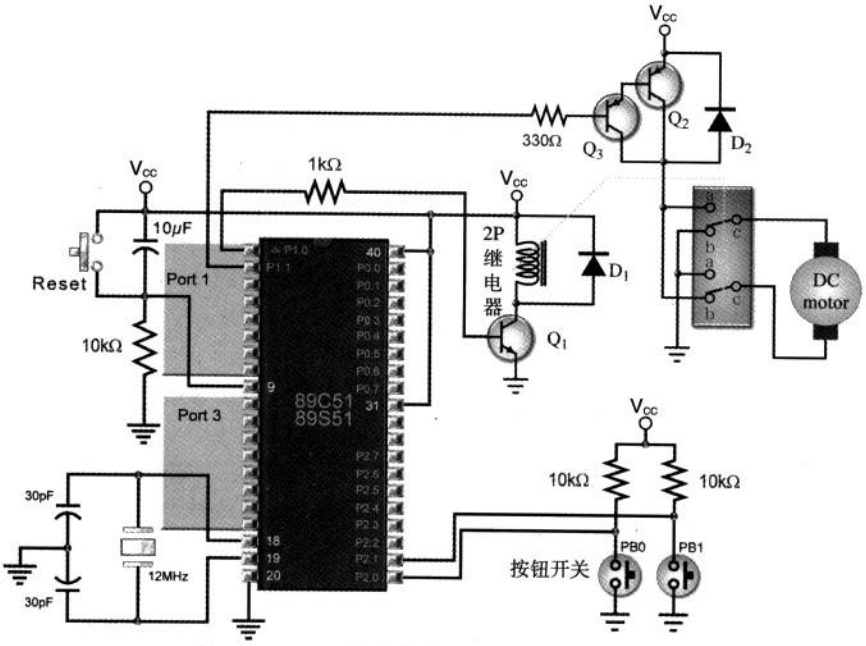
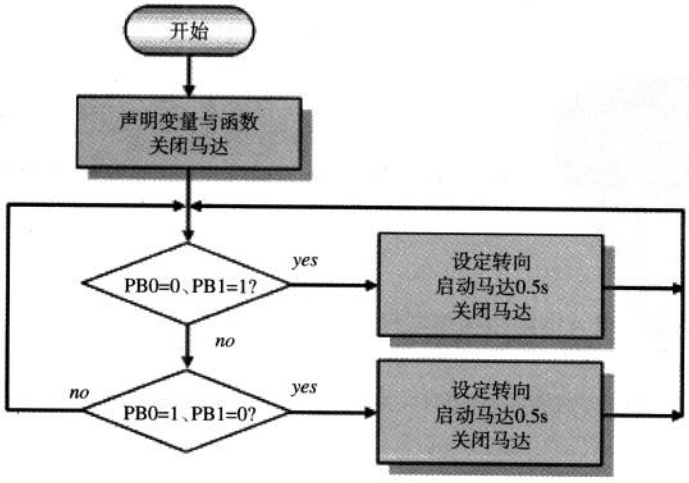


图14-22 继电器方向控制实验电路图

流程图与程序设计



```
/* 继电器方向控制实验(ch14-2.c)*/
#include <reg51.h>
sbit SW = P1^1;                          // 声明直流马达开关位置
sbit DIR = P1^0;                         // 声明直流马达转向位置
sbit PB0 = P2^0;                         // 声明PB0按钮开关位置
sbit PB1 = P2^1;                         // 声明PB1按钮开关位置
void delay1ms(int);                      // 声明延迟函数
//=========================================
main()
{ SW=1;                                  // 关闭直流马达
  PB0=1;                                 // 设定PB0
  PB1=1;                                 // 设定PB1
  while(1)                               // 无穷循环
  {     if(PB0==0 && PB1==1)             // 若按下PB0，且PB1未按下
        { DIR=1;                         // 设定直流马达转向
          SW=0;                          // 开启直流马达
          delay1ms(500);                 // 旋转0.5s
          DIR=0;                         // 恢复直流马达转向
          SW=1;                  }       // 关闭直流马达
        if(PB0==1 & PB1==0)              // 若按下PB1，且PB0未按下
        { DIR=0;                         // 设定直流马达转向
          SW=0;                          // 开启直流马达
          delay1ms(500);                 // 旋转0.5s
          SW=1;                  }       // 关闭直流马达
} }                                      // 结束
// ========== 延迟函数 =========
void delay1ms(int x)
{       int i,j;
  for(i=0;i<x;i++)                       // 外循环
       for(j=0;j<120;j++);               // 内循环
}                                        // 延迟函数结束
```

继电器方向控制实验（ch14-2.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若有错误或非预期的状况，

则检查原始程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图14-19连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

1. 若按住PB0不放，马达会怎样？
2. 试着修改本实验里的程序，让马达在放开按钮后才动作？

14.4.3 桥式方向控制实例演练

实验要点

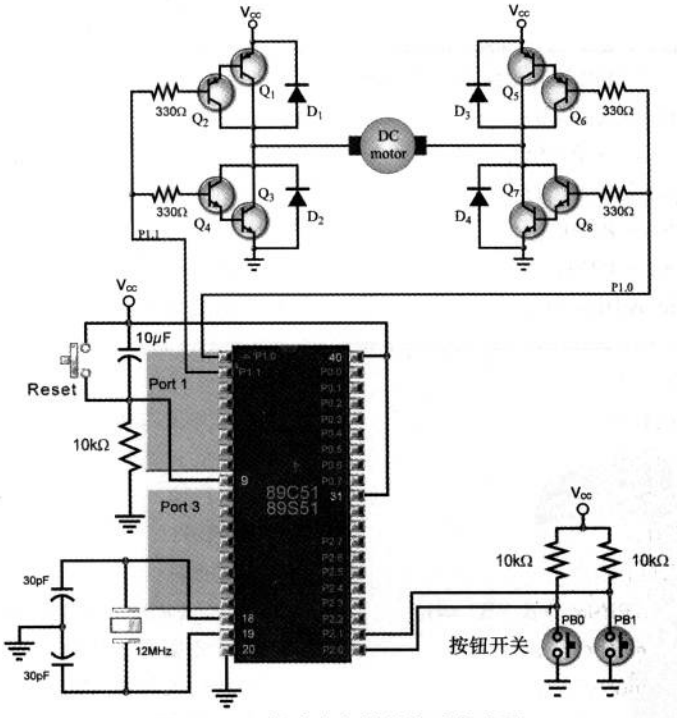
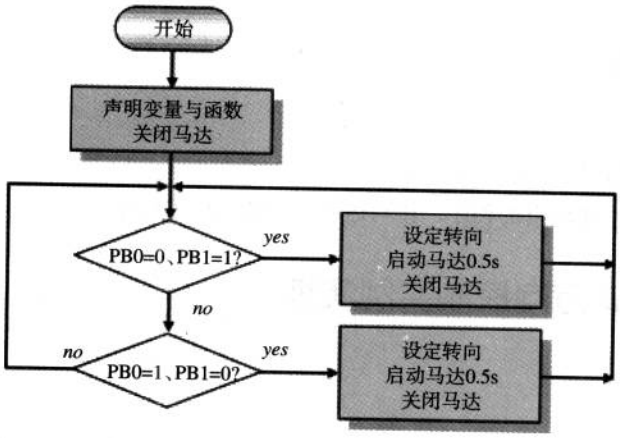


图14-23 桥式方向控制实验电路图

如图14-23所示，P1.0与P1.1连接到左右两组达林顿电路，假设P1.0=0，P1.1=1，马达顺时针转动；P1.0=1，P1.1=0，马达逆时针转动。P1.0=1，P1.1=1或P1.0=0，P1.1=0，马达皆不动作。按一下PB0钮，马达顺时针旋转0.5s；按一下PB1钮，马达逆时针旋转0.5s。若同时按住PB0与PB1钮，马达将不转动。

● 流程图与程序设计




```
/* 桥式方向控制实验(ch14-3.c) */
#include <reg51.h>
sbit motor1 = P1^0;                    // 声明直流马达位置
sbit motor2 = P1^1;                    // 声明直流马达位置
sbit PB0 = P2^0;                       // 声明按钮开关位置
sbit PB1 = P2^1;                       // 声明按钮开关位置
void delay1ms(int);                    // 声明延迟函数
//=========================================
main()
{ motor1=0;                            // 关闭直流马达
  motor2=0;                            // 关闭直流马达
  PB0=1;                               // 设定PB0
  PB1=1;                               // 设定PB1
  while(1)                             // 无穷循环
  {     if(PB0==0 & PB1==1)            // 若按下PB0，且PB1未按下
        {motor1=0;                     // 设定直流马达转向
         motor2=1;                     // 开启直流马达转向
         delay1ms(500);                // 旋转0.5s
         motor2=0;}                    // 关闭直流马达
        if(PB0==1 & PB1==0)            // 若按下PB1，且PB0未按下
```

```
        {motor1=1;                          // 设定直流马达转向
         motor2=0;                          // 开启直流马达转向
         delay1ms(500);                     // 旋转0.5s
         motor1=0;}                         // 关闭直流马达
} }                                         // 结束
// ========== 延迟函数 =========
void delay1ms(int x)
{       int i,j;
  for(i=0;i<x;i++)                          // 外循环
        for(j=0;j<120;j++);                 // 内循环
}                                           // 延迟函数结束
```

桥式方向控制实验（ch14-3.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若有错误或非预期的状况，则检查原始程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图14-20连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

请利用TA7257P替代图14-23中的4个达林顿晶体管，并重新进行实验。

14.4.4 PWM控制实例演练

实验要点

如图14-24所示，P1.0输出到达林顿晶体管，以控制直流马达。另外，8P的指拨开关连接到Port 2，由开关的状态决定P1.0所输出脉冲的工作周期，如表14-5所示，其中S8具有最高优先等级，随后依次为S7，S6……

表14-5

指拨开关	S1	S2	S3	S4	S5	S6	S7	S8
工作周期	12%	24%	36%	48%	60%	72%	84%	96%

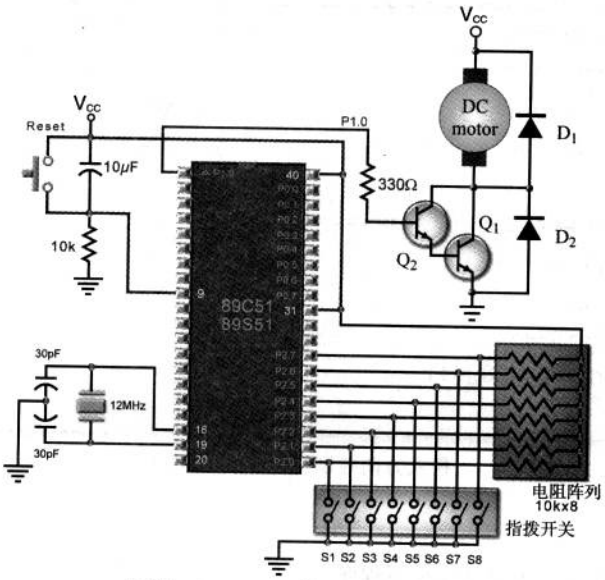
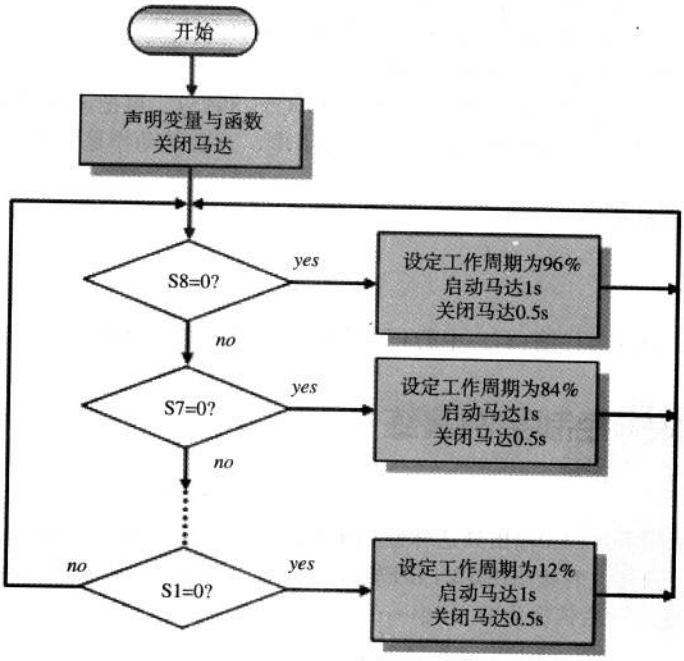


图14-24 PWM控制实验电路图

流程图与程序设计




```
/* PWM控制实验(ch14-4.c) */
#include <reg51.h>
```

```
sbit motor = P1^0;                    // 声明直流马达位置
sbit S8 = P2^7;                       // 声明指拨开关S8位置
sbit S7 = P2^6;                       // 声明指拨开关S7位置
sbit S6 = P2^5;                       // 声明指拨开关S6位置
sbit S5 = P2^4;                       // 声明指拨开关S5位置
sbit S4 = P2^3;                       // 声明指拨开关S4位置
sbit S3 = P2^2;                       // 声明指拨开关S3位置
sbit S2 = P2^1;                       // 声明指拨开关S2位置
sbit S1 = P2^0;                       // 声明指拨开关S1位置
void delay1ms(int);                   // 声明延迟函数
void output(char);                    // 声明输出函数
//=======================================
main()
{ motor=0;                            // 关闭直流马达
  P2=0xff;                            // 将P2规划为输入口
  while(1)                            // 无穷循环
  {     if(S8==0)                     // 按下S8
          output(96);                 // 指定工作周期
        else if(S7==0)                // 按下S7
          output(84);                 // 指定工作周期
        else if(S6==0)                // 按下S6
          output(72);                 // 指定工作周期
        else if(S5==0)                // 按下S5
          output(60);                 // 指定工作周期
        else if(S4==0)                // 按下S4
          output(48);                 // 指定工作周期
        else if(S3==0)                // 按下S3
          output(36);                 // 指定工作周期
        else if(S2==0)                // 按下S2
          output(24);                 // 指定工作周期
        else if(S1==0)                // 按下S1
          output(12);                 // 指定工作周期
  }
}                                     // 结束
// ========== 输出函数 =========
void output(char on)
{       char i;
  for(i=0;i<10;i++)                   // 循环
```

```
        { motor=1;              // 输出高电平
          delay1ms(on);         // 延迟on时间
          motor=0;              // 输出低电平
          delay1ms(100-on);     // 延迟100-on时间
        }                       // 结束
  delay1ms(500);                // 马达停止0.5s
}                               // 延迟函数结束
// ========== 延迟函数 =========
void delay1ms(int x)
{       int i,j;
  for(i=0;i<x;i++)              // 外循环
        for(j=0;j<120;j++);     // 内循环
}                               // 延迟函数结束
```

PWM控制实验（ch14-4.c）

操作

1. 根据功能需求与电路结构编写程序，然后将该程序编译与链接，以生成*.HEX文件。

2. 在Keil C里进行软件调试/仿真，看看其功能是否正常？若出现错误或非预期的状况，则检查原始程序，看看哪里出了问题？并将它记录在实验报告里。

3. 若软件调试/仿真功能正常，再按图14-21连接线路，使用在线仿真器，加载新的程序（*.HEX），以仿真该电路的动作。若出现非预期的状况，则检查线路的连接状况，看看哪里出了问题？并将它记录在实验报告里。

4. 若在线仿真功能正常，请将程序烧录到89C51/89S51，再把该89C51/89S51放入实体电路，以取代刚才的在线仿真器，然后直接上电，观察其动作是否符合预期？并记录在实验报告里。

5. 编写实验报告。

思考一下

请利用在本实验的电路与程序，找出当工作周期低于多少时，马达就不会转动？

14.5 实时练习



在本章中探讨了直流马达的结构、驱动方式与应用，在此请试着回答下列问题，以确认对于此部分的掌握程度。

选择题

() 1. 直流马达根据其绕组的连接方式可分为哪几种?
(A) 差分激式与和分激式 (B) 串激式与分激式
(C) 并激式与分激式 (D) 绕激式与直激式

() 2. 对于复激式马达，根据其连接方式可分为哪几种?
(A) 长复激式与短复激式 (B) 串复激式与分复激式
(C) 并复激式与分复激式 (D) 绕复激式与直复激式

() 3. 对于采用永久磁铁定子的直流马达，若要改变其转向，应如何处理?
(A) 改变场绕组的电流方向 (B) 将电源反接
(C) 将电源反接，并将转子的电源反接 (D) 使用交流电

() 4. 达林顿晶体管的特色是什么?
(A) 反应速度快 (B) 耐压高 (C) 高电流增益 (D) 高电压增益

() 5. TIP12x系列所提供的电流(I_C)可达多少A?
(A) 60A (B) 30A (C) 10A (D) 5A

() 6. TIP14x系列所提供的电流(I_C)可达多少A?
(A) 60A (B) 30A (C) 10A (D) 5A

() 7. 若要使用继电器控直流马达的转向，应采用何种继电器?
(A) 耐高压型 (B) 至少为2P型 (C) 采用一般1 P型即可 (D) 高电流型

() 8. 所谓“PWM”是指什么调变?
(A) 脉冲编码调变 (B) 脉冲幅度调变
(C) 脉冲频率调变 (D) 脉冲宽度调变

() 9. PDM是以改变什么来调整功率的?
(A) 工作周期 (B) 起动时间 (C) 脉冲幅度 (D) 延迟时间

() 10. PDM与下列哪项相同?
(A) PAM (B) PCM (C) PWM (D) PPM

简答题

1. 试绘图以说明桥式控制直流马达转向的动作。

2. 若要利用桥式控制直流马达的转向，而此直流马达的电流约为6A，可采用哪组达林顿晶体管配对?

3. 请利用运算放大器，设计一个PWM波的调变电路。

4. 若要将PWM波形解调，变回正弦波，可使用什么电路?

5. 若要在8051里，以程序的方法产生PWM波，可应用什么函数?

() 3. 对于重载大起动，制动和正反向频繁的电动机，可以为减其体积。

(A) 串励式 (B) 并励式 (C) 他励式 (D) 复励式

() 4. 为了减小起动电流，对鼠笼式异步电动机可以采用降压起动。

(A) 定子串电阻 (B) 转子串电阻 (C) 转子串频敏 (D) 定子串电抗器

() 5. 对于要求起动转矩很大，机械特性软的负载，应选用 电动机。

(A) 他励直流 (B) 并励直流 (C) 串励直流 (D) 三相鼠笼异步

() 6. 三相异步电动机的调速方法有 。

(A) 变极调速 (B) 变频调速 (C) 变转差率调速 (D) 以上都是

() 7. 为了减小起动电流，对容量较大的直流电动机应采用 起动。

(A) 全压 (B) 降压 (C) 转子串电阻 (D) 以上都可以

() 8. 三相异步电动机的调速方法有 。

(A) 变极调速 (B) 变频调速 (C) 变转差率调速 (D) 以上都是

() 9. 三相异步电动机的调速方法有 。

(A) 变极调速 (B) 变频调速 (C) 变转差率调速 (D) 以上都是

() 10. PWM 表示 。

(A) PAM (B) PCM (C) PWM (D) PPM

二、填空题

1. 直流电动机的起动方法有 和 两种。

2. 异步电动机降压起动是为了减小起动电流，但同时也减小了 ，所以此种起动方法适用于 的场合。

3. 异步电动机变频调速时，变频电源应具有 和 的功能。

4. 脉宽调制 PWM 的英文缩写是 。

5. 三相异步电动机的调速方法有 、 和 。

三、简答题

1. 直流电动机为什么不能直接起动？起动电流太大有什么危害？

2. 三相异步电动机的起动方法有哪些？各适用于什么场合？

3. 三相异步电动机的调速方法有哪些？各适用于什么场合？

4. 三相异步电动机的制动方法有哪些？各适用于什么场合？

5. 三相异步电动机的调速方法有哪些？各适用于什么场合？

附录A 开发工具简介

本章内容丰富，主要包括以下4部分。

WINICE-51/52E在线仿真器：

介绍WINICE-51/52E软硬件的安装与应用。

WICE-52在线仿真器：

介绍WICE-52软硬件的安装与应用。

PICE-52在线仿真器：

介绍PICE-52软硬件的安装与应用。

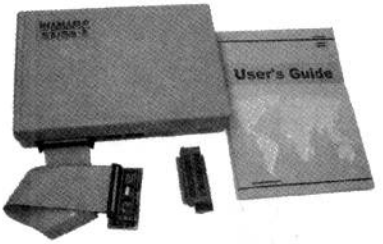
89S51在线烧写实验板：

介绍在线烧写实验板，包括完整电路图、所用器件以及驱动程序的操作等。

A.1 WINICE-51/52E在线仿真器



新华计算机股份有限公司（公司网址为www.microtime.com.tw）可以说是开发在线仿真器的先锋，该公司开发的WINICE-51/52E在线仿真器具有技术成熟、性能稳定的特点。WINICE-51/52E在线仿真器除了提供在线仿真功能，还能进行89C51系列的烧写与验证，也就是说，有了它连**89C51**的烧写器也可以省了！而且该在线仿真器附有一本非常详实的操作手册，公司也提供了良好的售后服务。



WINICE-51/52E在线仿真器

规格介绍

以下WINICE-51/52E在线仿真器的规格取自新华计算机股份有限公司网站。

- 可仿真的CPU包括80（C）31/32、80（C）51/52、87（C）51/52等，且支持5V 89C51/52烧写功能。
- 仿真频率可达16MHz（以软件切换内外部CLOCK）。
- 提供128KB仿真存储器（program 64KB、data 64KB）。
- 采用并口（PRINTER PORT）通信。
- 提供64KB硬件执行断点及硬件总线断点可设定：Address、Data bus、CPU Status及4 External Trigger。
- 可实时记录8k × 32bits的数据，包括16 bit address、8 bit data、4bit status及4 bit external probes。
- 追踪深度可调整为Forward、Backward 及About Trace及提供追踪设定点及Qualify Trace。
- 追踪后的结果可以针对上述32 bits中任选16 channels作波形显示（Waveform Display）。
- 提供下载（Download）文件，支持HEX、BIN、OMF等12种格式。
- 具有Batch及Log处理能力，并可进行在线（On Line）全屏幕行编译及反编译。
- 具有符号调试（Symbolic Debug）能力。
- 可扩展MHLD（Microtime High Level Debugger），它为使用者提供了高阶调试环境（可针对C语言）。
- 提供三种程序显示模式：纯高级语言、高级语言与汇编语言混合以及纯汇编语言。

硬件的连接

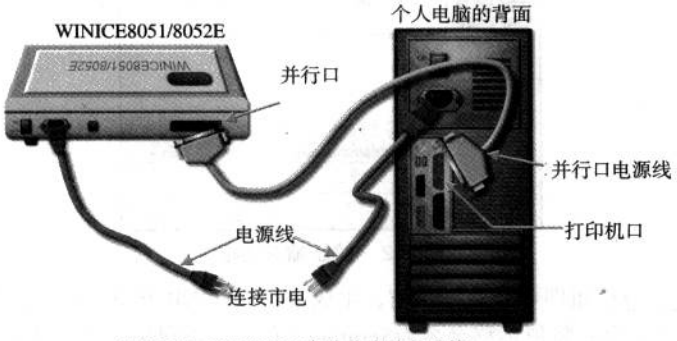




WINICE-8051/8052E背面图

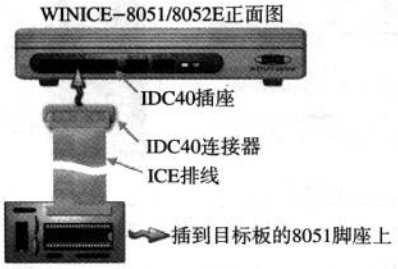
在使用WINICE-51/52E在线仿真器之前，首先要进行设备的连接，其中包括三个步骤。第一步是连接WINICE-51/52E在线仿真器的电源，使用DC5V直流电源时，可使用厂方提供的连接计算机键盘的连接线，直接从键盘上接电源。不过，笔者还是认为用市电比较好，将市电变压后连接到WINICE-51/52E在线仿真器的AC110V电源孔即可，方便又简单。

第二步是使用附带数据线把WINICE-51/52E在线仿真器与计算机的打印机端口连接起来。原本是件简单的事，但WINICE-51/52E在线仿真器只能与EPP模式的打印机输出端口相连。若无法顺利联机，则需要重新开机，进入BIOS设置界面，将打印机端口设定为EPP模式。不同品牌主板的BIOS设置方法大同小异，具体方法可查阅主板说明书，在此不作说明。



WINICE-8051/8052E与个人电脑的连接

第三步是连接WINICE-51/52E在线仿真器和目标板（target board），实际上，在线仿真器的功能就是仿真8051CPU的功能，所以要用排线将仿真器与目标板的8051底座相连。



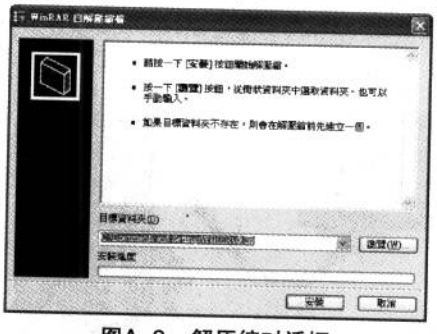
软件安装

在**WINICE-8051/8052E**在线仿真器的包装中附有一张驱动软件的光盘，驱动软件也可直接从该公司网站下载，网址是“www.microtime.com.tw/drivers/winicew.htm”，图A-1为驱动软件的下载页面。



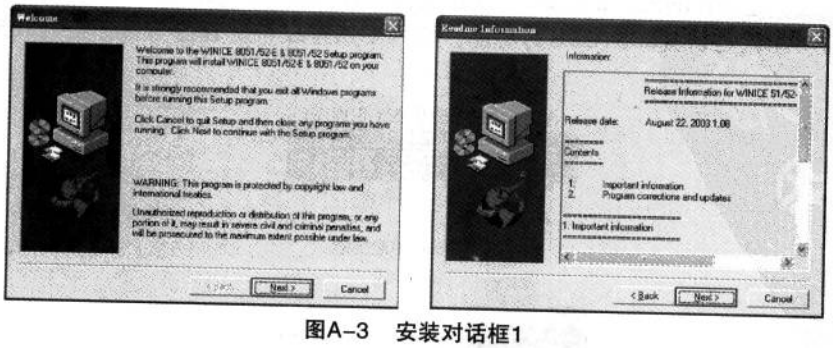
图A-1 下载页面

下载的文件为压缩文件（约1.8MKB），下载完成后，将鼠标移至该文件的图标，双击鼠标左键，打开如图A-2所示的解压缩对话框。



图A-2 解压缩对话框

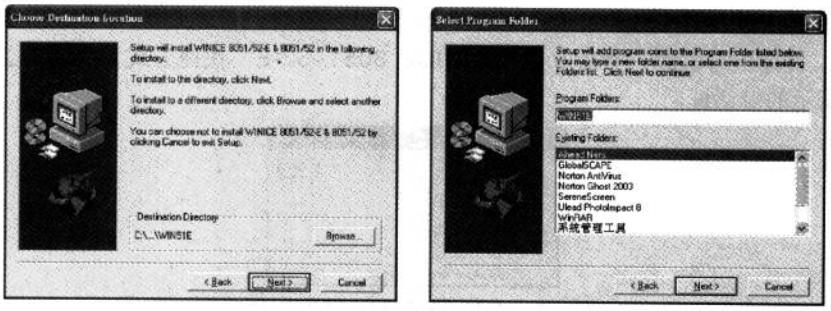
单击 安装 按钮即可开始解压缩，生成一个“WINICE 51_52_E（V3.06）”文件夹。在这个文件夹里，将鼠标移至名为Setup的文件，双击鼠标左键，开始安装驱动软件，屏幕上出现如图A-3左图所示的安装对话框。



图A-3 安装对话框1

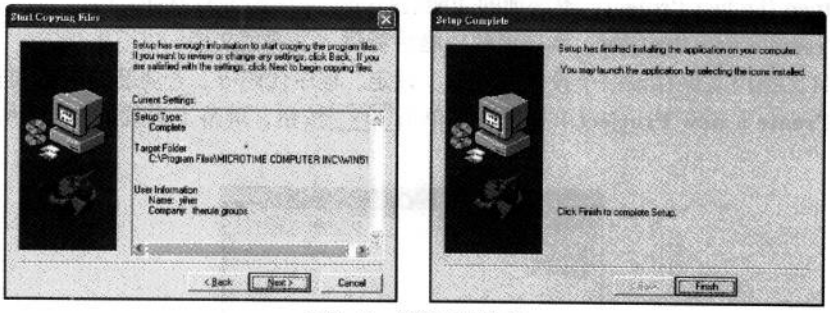
单击 Next > 按钮切换到下一个对话框，如图A-3右图所示；再单击 Next > 按钮切

换到下一个对话框，如图A-4左图所示。



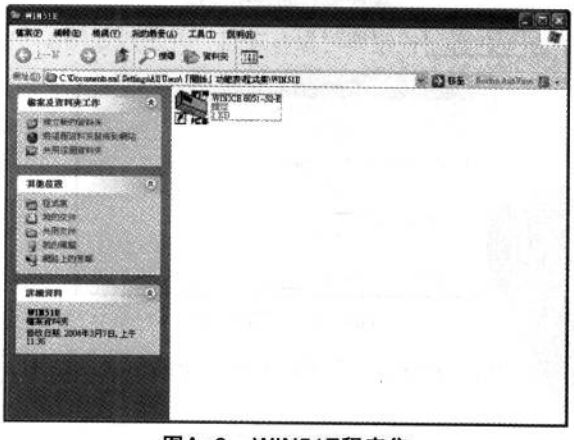
图A-4 安装对话框2

若不使用默认安装路径，则单击 Next > 按钮，如图A-4右图所示；再单击 Next > 按钮，切换到下一个对话框，如图A-5左图所示。



图A-5 安装对话框3

单击 Next > 按钮即可切换到下一个对话框，如图A-5右图所示；最后单击 Next > 按钮完成安装，并产生一个WIN51E程序集，如图A-6所示。

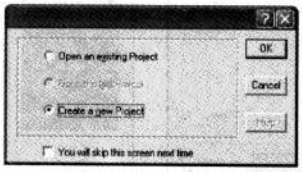


图A-6 WIN51E程序集

其中只有一个WINICE 8051~52-E的快捷方式，双击这个快捷方式可以运行该驱动软件。

运行方法

当需要使用WINICE-8051/8052E软件进行在线仿真时，首先要将WINICE分别连接到计算机、目标板与电源，然后双击“WINICE 8051~52-E”快捷方式，屏幕上会出现如图A-7所示的对话框。

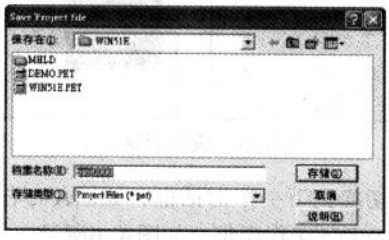


图A-7 WINICE 8051/8052E对话框

对话框中有如下三个选项。

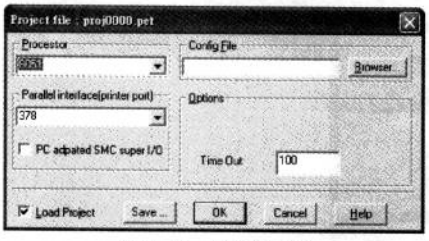
- **Open an existing Project**：本选项的功能是打开一个已有的项目文档。
- **Open the last Project**：本选项的功能是打开上一次编辑的项目文档。
- **Create a new Protject**：本选项的功能是新建一个项目文档。

项目文档用来保存该一个设计的参数、设定、程序代码等，若要新建一个项目文档，则选择**Create a new Protject**单选钮，再单击 OK 按钮，屏幕上出现如图A-8所示的对话框。



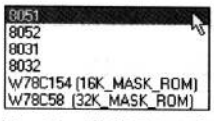
图A-8 储存项目档

选定文件名，单击 储存(S) 按钮即可保存项目，并出现如图A-9所示的对话框。



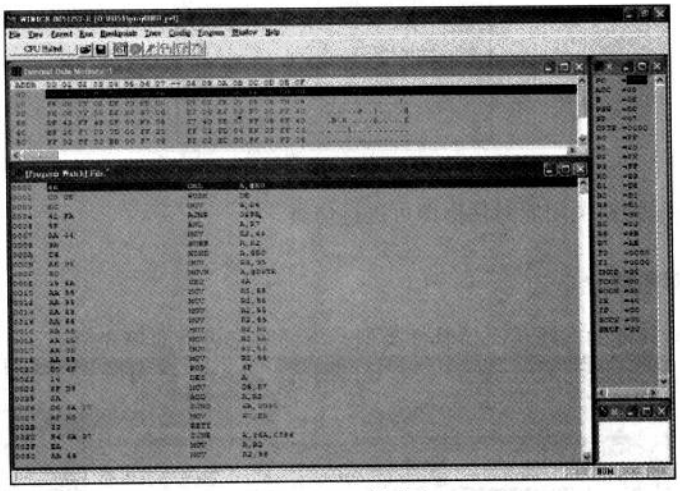
图A-9 设定环境

可在对话框左上方的**Processor**下拉列表中选择所要仿真的CPU，其中包含的选项如图A-10所示。



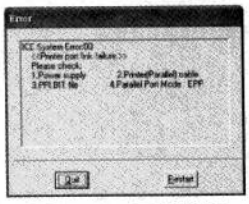
图A-10 仿真CPU列表

选定后单击 OK 按钮，进入WINICE的主界面，如图A-11所示。



图A-11 主界面

若连接不成功或打印机模式设定不正确，将出现如图A-12所示的对话框。

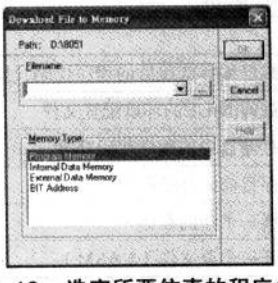


图A-12 无法连接

先单击 Quit 按钮关闭此对话框，检查线路连接是否正确。问题一般是出在打印机模式设定不正确，只要重新开机，进入BIOS设置界面，将计算机的打印机模式设定为EPP模式即可。

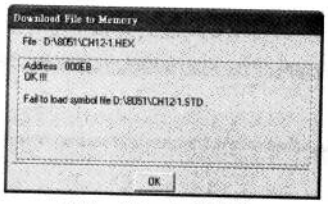
加载程序代码

当WINICE与计算机顺利连接后，即可载入已编译的文件（.hex）。运行**File/Open Code**命令，出现如图A-13所示的对话框。



图A-13 选定所要仿真的程序代码

在Filename下拉列表中选择所要仿真的文件（*.hex）路径，也可单击 ... 按钮查找。选定文件后，单击 OK 按钮，即可加载该文件，出现如图A-14所示的对话框。

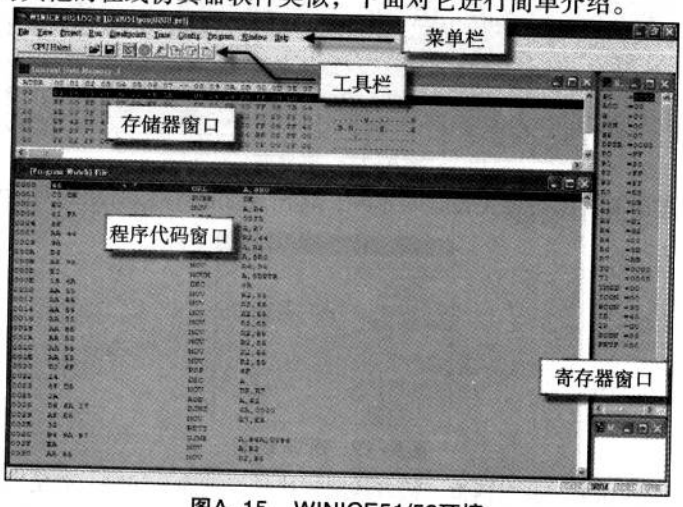


图A-14　加载程序

再单击 OK 按钮关闭此对话框即可进行仿真。

WINICE环境简介

WINICE与其他的在线仿真器软件类似，下面对它进行简单介绍。



图A-15　WINICE51/52环境

菜单栏

在WINICE的菜单栏里提供了10个菜单，这10个菜单之中有许多命令，如表A-1所示。

表A-1

命令	功能说明
File菜单	
Open Code...	加载已编译链接后的程序代码，如*.hex文件
Open ICE Configuration...	加载WINICE的环境配置文件
Save Code as...	将存储器中的程序代码存盘
Save ICE Configuration...	保存目前的环境配置
New	编辑一个新的程序
Open Source...	打开一个源文件（*.ASM）
Save Source	储存目前所编辑的程序源文件
Save Source As...	将目前所编辑的程序源码另存为新文件
Exit	退出程序

续表

命 令	功 能 说 明
View菜单	
Add a Watch point	新增参数观察点
Message	将所有动作列表记录
Project菜单	
New	打开新项目文件
Open	打开项目文件
Modify	修改目前的项目文件
Run菜单	
Free Running	全速仿真
Auto Step over Auto Step into	自动单步仿真
Step over	单步仿真（如遇到子程序，则直接执行完该子程序）
Step into	单步仿真
Go to cursor	仿真到光标所在位置
Execute to...	仿真到指定位置
Stop ICE	停止仿真
Reset Target MCU	复位目标板
Reset and Go	复位后全速仿真
Breakpoint菜单	
Toggle S/W Break	软件断点切换
H/W Break Setup...	硬件断点设定
S/W Break Setup...	软件断点设定
Enable global breaks	启用整体断点
Disable global breaks	停用整体断点
Trace菜单	
Trace Setup...	设定追踪
Run & Trace auto stop	全速执行并追踪自动停止
Open new Trace window	开新追踪窗口
Config菜单	
CPU Control PINS Setting...	CPU控制接脚设定
Mapping Emulation Memory...	映对仿真存储器
CPU Pin Out	展示CPU管脚图
89C51/52 PROGRAM	烧写89C51/52
89C51 VERIFY	验证89C51/52内容
Color Setup...	设定颜色
Program*菜单	
Go to caret	程序执行到光标所在地址
Address（Go To...）	程序执行到指定地址

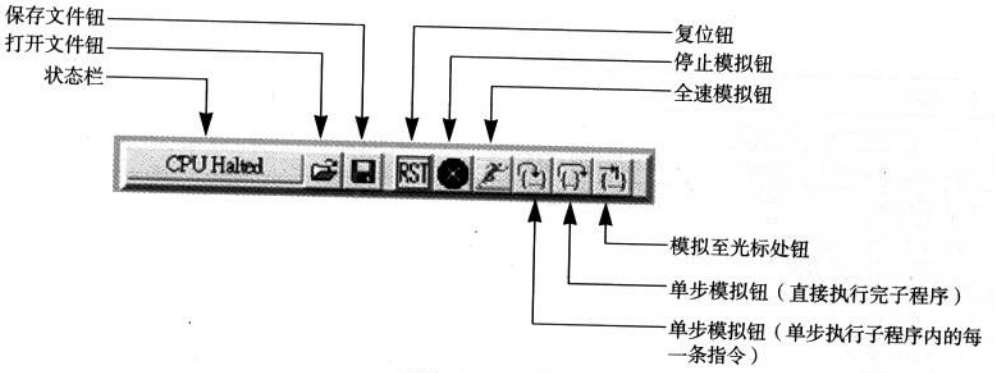
续表

命令	功能说明
New PC display at caret	将程序计数器PC所指的地址改为光标所在地址
Origin（Scroll to PC）	回到原点（程序计数器PC所指的地址）
Toggle S/W Break	切换软件断点
Source Mode（ASM/MIX）	源程序代码显示模式切换
Block Utilities	区域工具设定
Dump to File...	将存储器里的程序代码存盘
Window菜单	
Open new Program window	打开新的程序代码窗口
Open new Data window	打开新的数据窗口
Open new Register window	打开新的寄存器窗口
Open Watch point	打开参数观察窗口
Tile Horizontal Windows	水平排列显示各窗口
Tile Vertical Windows	垂直排列显示各窗口
Cascade Windows	重叠显示各窗口
Arrange Icons	重新排列窗口缩小图标
Close All	关闭所有窗口
Help菜单	
Contens	应用目录与索引，以查询相关说明
Using help	使用帮助
About\	显示WINICE程序版本

注：Program*表示该菜单会随光标而定，当光标在存储器窗口、程序代码窗口、寄存器窗口或消息窗口时，此菜单将显示该窗口的状态与命令。

• 工具栏

菜单太复杂，而使用WINICE的工具栏可以使操作更简单便捷。WINICE的工具栏包括一个状态区及8个快捷按钮，如图A-16所示。



图A-16 工具栏

• 存储器窗口

存储器窗口显示CPU的存储器状况，我们可从存储器的变化了解程序执行的状况。

• 程序代码窗口

程序代码窗口显示所加载的程序代码，其中包括编译后的代码与反编译程序（源程序）。

• 寄存器窗口

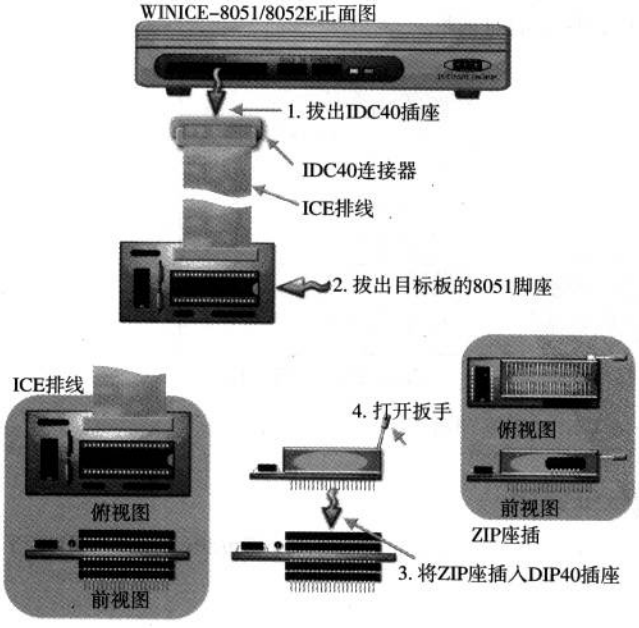
寄存器窗口显示CPU里各寄存器的状况，帮助使用者了解CPU及程序执行的状况。

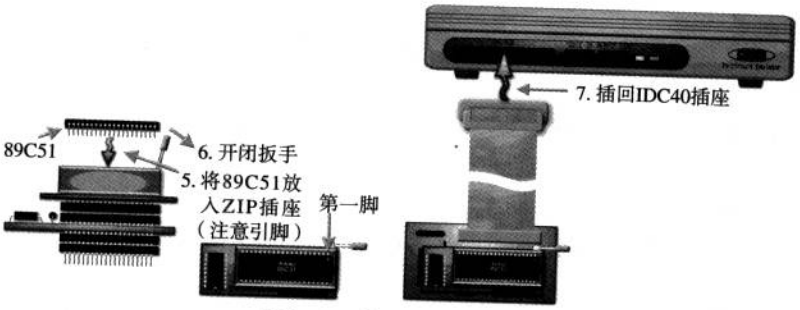
基本仿真实例

运行WINICE之后可以加载要仿真的程序代码，单击按钮（或运行菜单**File/Open Code...**命令），然后在随即出现如图A-28所示的对话框中选择所要仿真的文件（*.hex），并载入该文件。一般来说，开始仿真之前，要首先确定线路连接以及目标板没有问题；接着，单击RST按钮复位系统（确定从头开始），再单击按钮即可全速仿真，而目标板上的电路也随之工作，这时候，就可观察目标板电路的工作是否正确。另外，当我们单击按钮后，此按钮将变成，若要停止仿真，则单击按钮即可。

89C51烧写实例

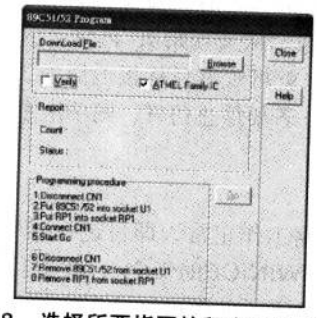
仿真完成，确定目标板电路工作正常，即可进行89C51的烧写。首先将ICE排线从WINICE端的接头中拔出（为了WINICE的安全），然后插到目标板的40Pin DIP插头上，拔出目标板的8051CPU，由于是不关电源进行操作的，要非常小心！然后再很小心地把烧写用的ZIP底座板插到刚才拔出目标板的上方，管脚一定要插对！最后把89C51放入ZIP底座，并锁住ZIP底座，再把ICE排线在WINICE端的接头插入WINICE，如图A-17所示。





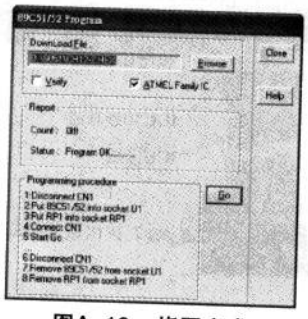
图A-17 烧写设备的连接

接着，运行**Config/89C51/52 PROGRAM...**命令，屏幕上出现如图A-18所示的对话框。



图A-18 选择所要烧写的程序文件（*.hex）

在**DownLoad File**下列列表中选择文件（也可按 Browse 按钮查找），再单击 Go 按钮即可载入该文件，即进行烧写的动作，这时候千万不要断电！很快程序就会烧写完成，屏幕上出现如图A-19所示的对话框。

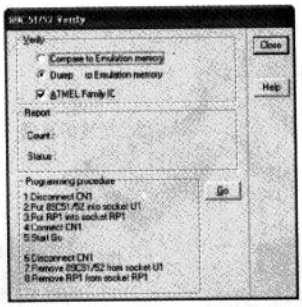


图A-19 烧写完成

单击 Close 按钮关闭对话框即可。

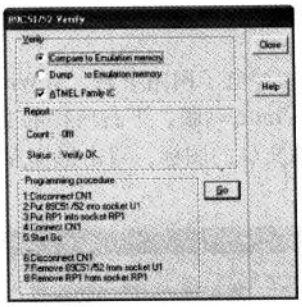
烧写数据的验证实例

完成烧写后，若要确认烧写无误，则运行**Config/89C51/52 VERIFY...**命令，屏幕上出现如图A-20所示的对话框。



图A-20 烧写数据验证对话框

选择**Compare to Emulation memory**选项，再单击 Go 按钮即可进行比对验证。很快即可完成验证，屏幕上出现如图A-21所示的对话框。



图A-21 完成数据比对

单击 Close 按钮关闭对话框即可。如果在烧写时激活了对话框中的Verify选项，则可在烧写完成后自动进行数据验证。如果使用传统的烧写器，还必须先将CPU内部的数据清零，才能烧写数据。而使用WINICE的烧写功能，就不必如此，不管有没有数据，直接烧写即可！

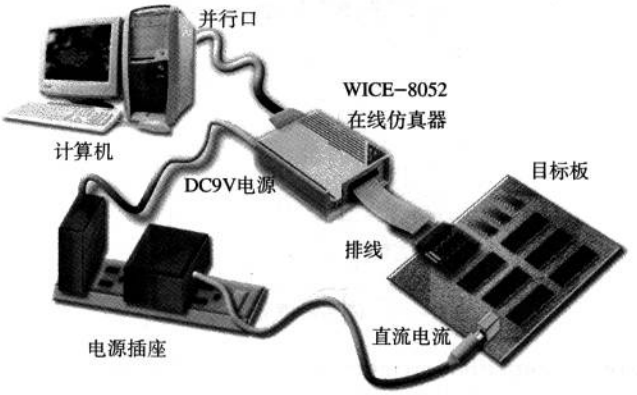
A.2 WICE-52在线仿真器



下面将介绍力浦电子公司（公司网址为www.leap.com.tw）所开发的WICE-8052在线仿真器，WICE-8052在线仿真器的相关资料与驱动程序。

● 硬件连接

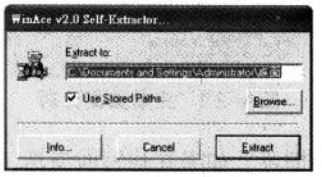
连接WICE-8052在线仿真器分为三个步骤，第一步是将WICE-8052在线仿真器附带的9V电源适配器（adapter），连到电源插座，另一头连接到WICE-8052的9V电源孔；第二步是使用数据线把WICE-8052在线仿真器和计算机的打印机端口连接；第三步是把WICE-8052在线仿真器附带的40pin排线插到目标板（target board）的8051底座上，如图A-22所示。



图A-22 WICE-8052在线仿真器的连接

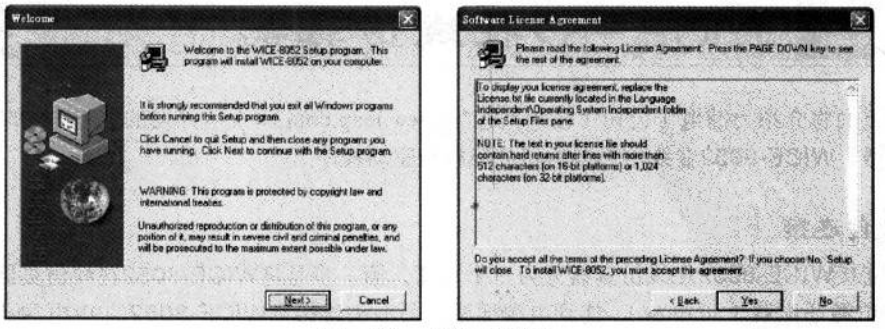
软件安装

WICE-8052的软件可以从力浦电子网站下载，有Windows版和DOS版两种版本，可根据需要下载。在此建议使用Windows版，下载所得的文件是一个压缩文件（文件名为Wice8052v217），将鼠标移至该文件的图标，双击鼠标左键，即可打如图A-23所示的对话框。



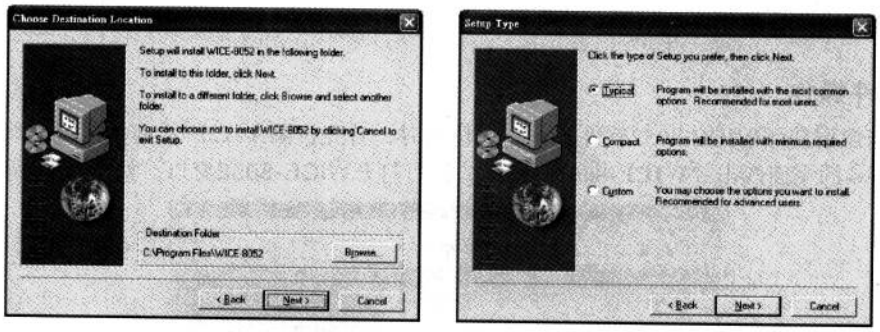
图A-23 解压缩窗口

单击 Extract 按钮即可自动解压缩，然后生成“Wice8052v217”文件夹，这个文件夹里有DISK1、DISK2两个文件夹，执行DISK1文件夹里的SETUP.EXE程序即可进行安装，屏幕上会显示如图A-24左图所示的对话框。



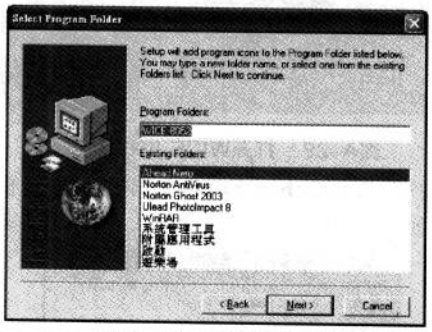
图A-24 安装对话框1

单击 Next > 按钮切换到下一个对话框，如图A-24右图所示；单击 Yes 按钮切换到下一个对话框，如图A-25左图所示。



图A-25 安装对话框2

如果使用默认安装路径，则单击 Next > 按钮，如图A-25右图所示；单击 Next > 按钮切换到下一个对话框，如图A-26所示。



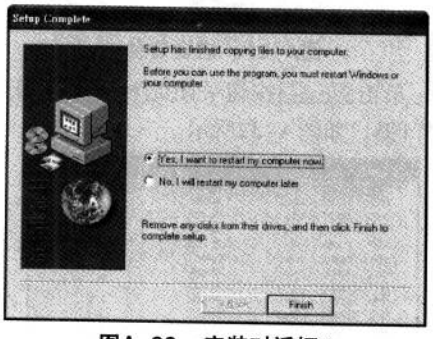
图A-26 安装对话框3

单击 Next > 按钮即可进行安装，屏幕上会出现如图A-27所示的对话框。



图A-27 确定对话框4

单击 确定 按钮切换到下一个对话框，如图A-28所示。

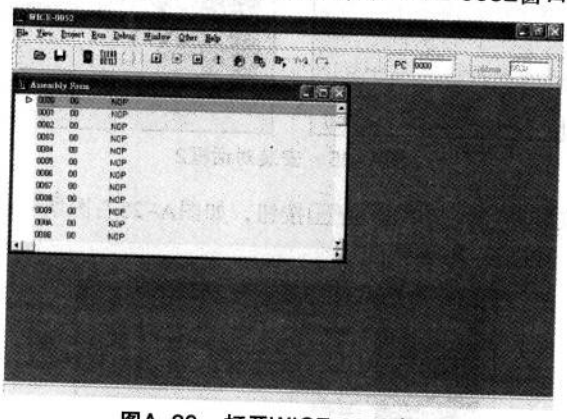


图A-28 安装对话框5

单击 Finish 按钮完成安装，并重新开机。

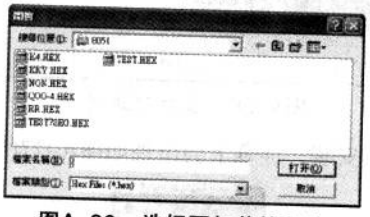
软件操作

当需要使用WICE-8052软件进行在线仿真时，首先按图A-22所示连接电路，然后在Windows的桌面双击“**WICE-8052**”图标，即可打开WICE-8052窗口，如图A-29所示。



图A-29　打开WICE-8052窗口

由于尚未加载程序，所以**Assembly Form**窗口中显示空白。若要加载程序，可单击 按钮，屏幕上会出现如图A-30所示的对话框。



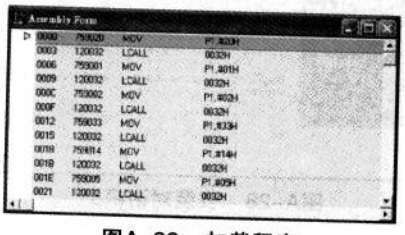
图A-30　选择要加载的程序

指定所要仿真的程序（*.hex），再单击 打开(O) 按钮，屏幕上会出现如图A-31所示的对话框。



图A-31　错误信息

忽略该对话框的内容，单击 确定 按钮关闭此对话框，即可加载该文档，而该文档将出现在Assembly Form窗口里，如图A-32所示。



图A-32　加载程序

若要进行仿真，则单击按钮或按 键，则目标板即开始工作。若要停止仿真，则单击按钮即可。

当然，WICE-8052在线仿真器还提供了许多其他功能，可利用Help/User Guide命令来参考其使用说明，在此不再赘述。

A.3 PICE-52在线仿真器



PICE-52在线仿真器是长高科技股份有限公司（公司网址为www.dmatek.com.tw）开发的8051/8052的在线仿真器。与前面两种在线仿真器相比，PICE-52在线仿真器有如下特点。

1. PICE-52在线仿真器与计算机之间采用串行端口接口（RS-232C）相连。

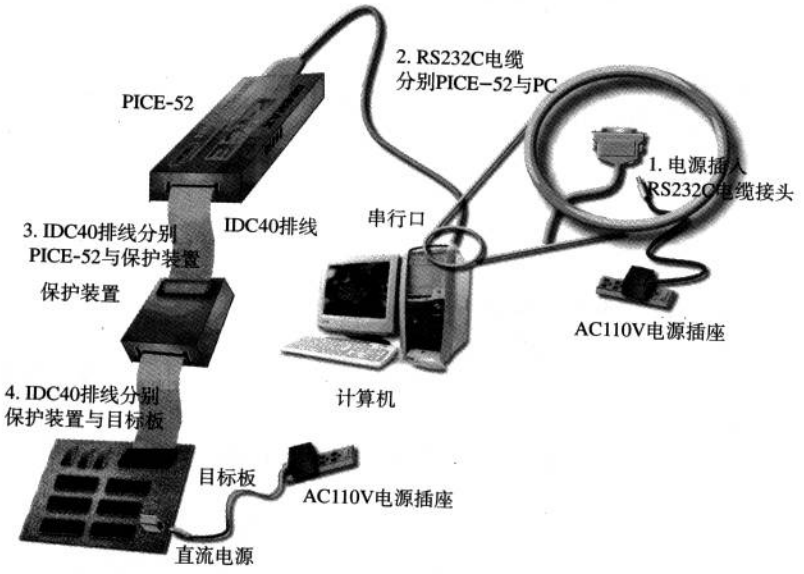
2. PICE-52在线仿真器提供全中文的在线使用说明，用户可直接在**Help**菜单里找到相关说明。

规格

- 实时仿真频率可达40MHz。
- 具有256KB的程序存储器及64KB的数据存储器。
- 可以把所有存储器设置为内部、外部、写入保护及不可擦写等各种组合的属性。
- 可以对内部、外部及CPU存储器进行浏览、检查、填充、寻找、比较、测试、合并、搬动等操作。
- 提供16k × 32bit的实时跟踪记录器，用以记录CPU的地址、数据、状态、外部信号等所有信号。
- 实时跟踪记录器可以显示机器码、汇编语言及CPU周期状态等数据。
- 实时跟踪记录器和过滤器结合可以过滤不想记录的数据，以确保数据的正确性。
- 提供256k个实时执行断点。
- 提供3个实时总线事件断点，每一断点又可为CPU地址、数据、状态、外部信号及事件发生次数的组合，且地址的设定方式可为区域、万用字符及特定值等多种设定。
- 提供3~5个外部信号中断输入。
- 提供2层实时事件激发，每一层又可分为OR/AND的组合。
- 具有全速执行、单步执行、单步呼叫执行及光标执行等方式。
- 提供输入、输出口检查。
- 提供智能型人机接口（WUIT-User Interface Tool For Windows），出色的人机接口可大幅缩短研发时间。
- 支持高级C语言调试环境（Keil C51 and IAR C51 compiler）。
- 具有十个多重窗口，支持随时自动更新正确数据，并会用不同颜色来提醒用户，哪些数据刚被改变。
- 更人性化的操作，不用执行命令也可轻易操作各项调试功能。

- 窗口可重叠（Overlay）、依靠（Docking）显示，各窗口间支持拖放功能、断点设置、浏览程序、观察变量或存储器内容等，只要移动鼠标即可完成。
- 支持变量/寄存器快速显示功能，把鼠标光标放在程序变量/寄存器上方即可显示其值。
- 强大的程序调试命令，不但可使工作自动化，还可设计自定义功能的工具按钮。
- 支持在线帮助说明，方便初学者使用。
- 可储存和恢复工作信息，方便用户进行持续的项目开发。

硬件连接



图A-33 PICE-52的连接

PICE-52与个人计算机（PC）之间通过RS-232C接口连接，而PICE-52的电源取自RS-232C上的电源插头，所以必须把电源供给器连接到RS-232C上的电源插头！如图A-33所示，首先把电源供给器的输出连接到PICE-52所附的RS-232C电缆接头（靠PC那端），然后再将该电缆分别连接到PICE-52和PC的串行口（COM1或COM2）；紧接着将PICE-52通过其所附的40pin排线连接其保护装置，再将保护装置的40pin排线连接到目标板（target board）的8051底座即可。

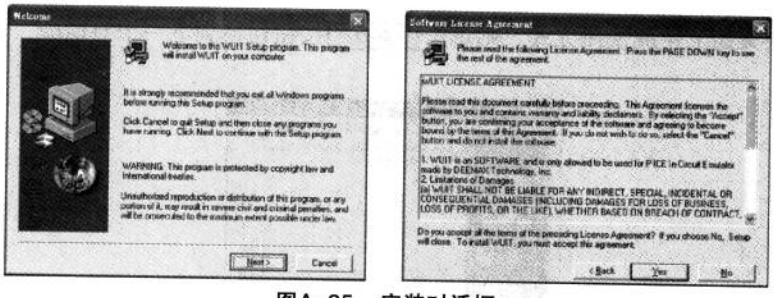
软件安装

PICE-52的软件可直接从网上下载，网址是“www.deemax.com.tw/ chinese.htm”，下载页面如图A-34所示。



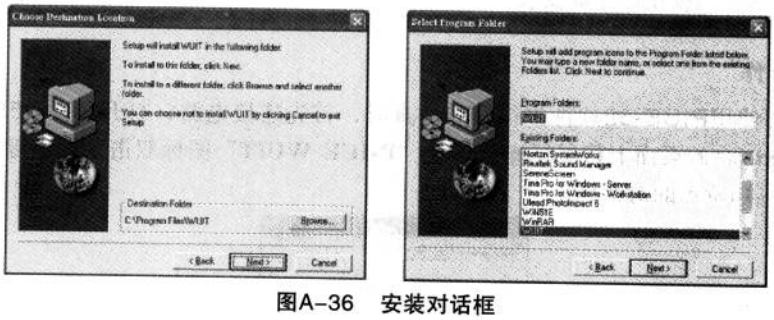
图A-34 下载页面

下载页面中有该软件的完整版、Windows版、DOS版和中文说明书，可根据需要下载，在此建议使用Windows版（占用的资源比较少）。下载所得的文件是一个压缩文件（Wuitsetup），首先将鼠标移至该文件的图标双击左键，将此文档解压缩（解压缩步骤省略），即可产生一个Wuitsetup文件夹，在此文件夹里，运行setup程序，即可运行安装程序，屏幕上出现如图A-35左图所示的对话框。



图A-35 安装对话框

单击 Next > 按钮切换到下一个对话框，如图A-35右图所示；单击 Yes 按钮切换到下一个对话框，如图A-36左图所示。



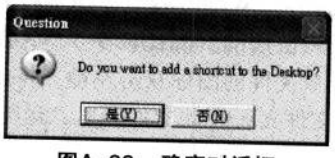
图A-36 安装对话框

如果不想变安装路径，则单击 Next > 按钮，如图A-36右图所示；单击 Next > 按钮切换到下一个对话框，如图A-37所示。



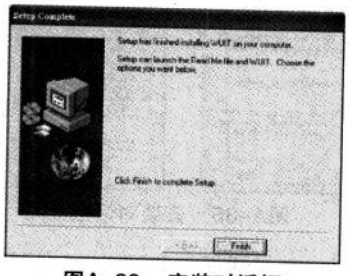
图A-37　安装对话框

在此选择采用哪一种语言的在线帮助，如果选择中文帮助，选择**Traditional Chinese（Taiwan）**选项，再单击 Next > 按钮，屏幕上出现如图A-38所示的对话框。



图A-38　确定对话框

若要在桌面上生成快捷方式，则单击 是(Y) 按钮切换到下一个对话框，如图A-39所示。

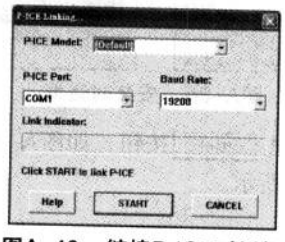


图A-39　安装对话框

单击 Finish 按钮即完成安装。

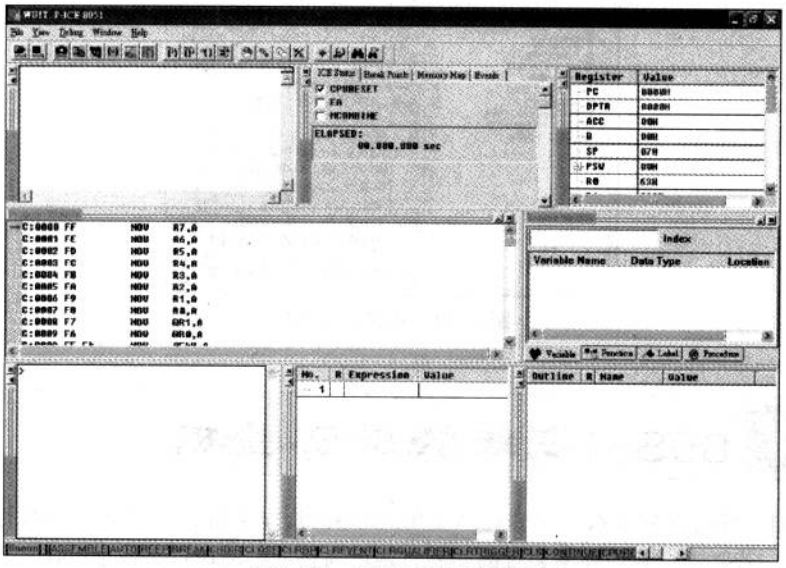
软件操作

当需要使用PICE-52软件进行在线仿真时，首先连接电路，打开PICE-52的开关，然后在Windows的桌面上将鼠标光标移至“**P-ICE WUIT**”图标双击鼠标左键，屏幕上出现如图A-40所示的对话框。



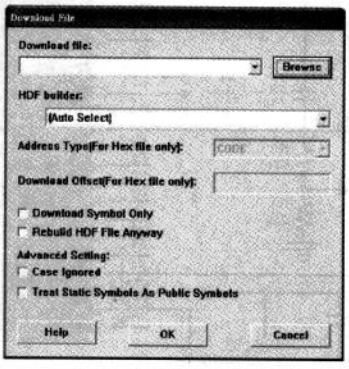
图A-40　链接P-ICE对话框

保持程序默认状态，单击 START 按钮，屏幕上出现如图A-41所示的窗口。



图A-41 打开PICE-52窗口

接着加载所要仿真的程序（*.hex），运行**Debug/Download**命令，屏幕上出现如图A-42所示的对话框。

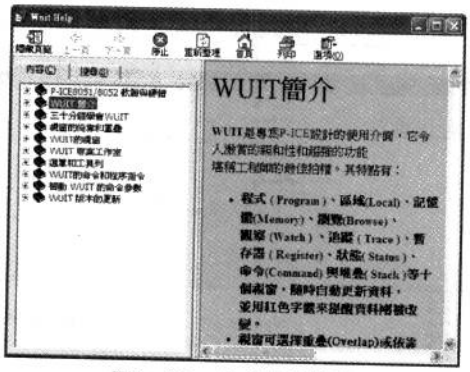


图A-42 加载程序代码对话框

单击 OK 按钮关闭此对话框，即可加载该文档。

若要进行仿真，则单击 按钮或按 F5 键，则目标板开始工作。若要停止仿真，则单击 按钮即可。

上述操作只是简单地进行仿真，PICE-52还提供许多其他功能，若有疑问或想进一步认识这套系统，可运行Help/Contents命令，参考其中文的使用说明，如图A-43所示。

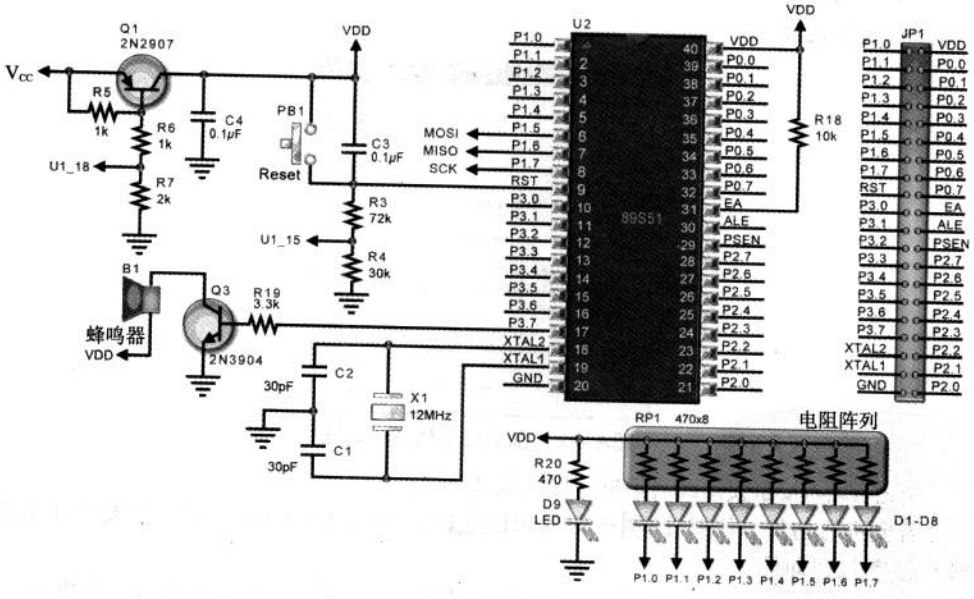


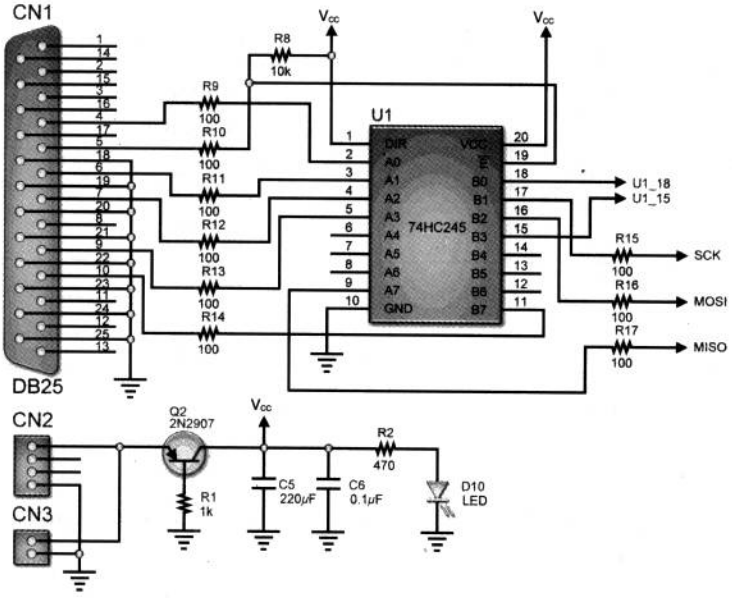
图A-43 中文使用说明

A.4 89S51在线烧录实验板



89S51在线烧录实验板是一套相当实用的89S51烧录装置，也可进行简单的LED与蜂鸣器实验。若要进行其他外围装置的实验，则可通过其中的JP1，将4个Port连接到外部的外围装置即可进一步进行实验。如图A-44所示为此实验板的完整电路图。





图A-44 89S51在线烧录实验板的电路

其中的器件如表A-2所示。

表A-2

零件序号	零件值	数量	说 明	备 注
U1	74HC245	1	含20pin脚座	
U2	89S51	1	含40pin脚座或省力脚座	
CN1	DB25	1	打印机端口连接器	
CN2		1	USB连接器	
CN3		1	两针脚电源连接器	
JP1		1	40pin双排针连接器	
Q1、Q2	2N2907	2	PNP晶体管	
Q3	2N3904	1	NPN晶体管	
X1	石英晶体	1	12MHz	
B1	蜂鸣器	1		
D1-D8	LED	8		
D9、D10	LED	2		
PB1	按钮开关	1		
RP1	电阻排	1	470Ω/8R9P	
R1、R5、R6	电阻	3	1kΩ/0.25W（或1/8W）	
R2、R20	电阻	2	470Ω/0.25W（或1/8W）	
R3	电阻	1	72kΩ/0.25W（或1/8W）	
R4	电阻	1	30kΩ/0.25W（或1/8W）	
R7	电阻	1	2kΩ/0.25W（或1/8W）	

续表

零件序号	零件值	数量	说 明	备 注
R8、R18	电阻	2	10kΩ/0.25W（或1/8W）	
R9～R17	电阻	9	100Ω/0.25W（或1/8W）	
R19	电阻	1	3.3kΩ/0.25W（或1/8W）	
C1、C2	电容	2	30pF	
C3、C4、C6	电容	3	0.1μF（104）	
C5	电容	1	220μF/25V	
	连接线	1	打印机端口连接线	
	连接线	1	USB连接线	
	驱动光盘	1		

89S51在线烧录实验板是由升利科技有限公司专为教学与个人学习/开发89S51系统所设计的。

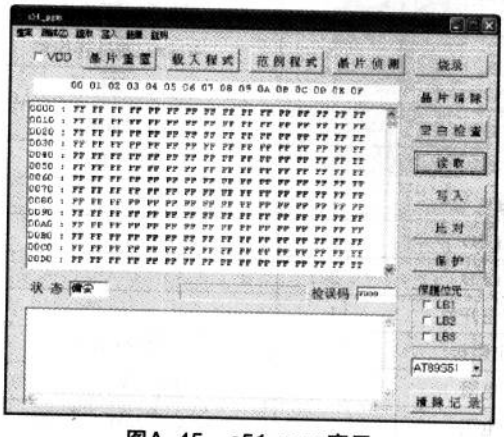
当需要应用89S51在线烧录实验板时，首先将所要烧路或实验的89S51放入U3脚座，此实验板采用+5V的电源，其引接方式有如下两种。

1. 直接利用USB数据线连接计算机，USB所提供的电源到实验板的CN2，即可取得+5V；同时可由CN3将电源引接到其他外围装置。当然，计算机所提供的+5V电源，绝对足够为89S51在线烧录实验板供电，而通过CN3引接到其他外围装置时，负载不能过大，以免引起计算机供电不足。

2. 由+5V直流电源供给器连接到89S51在线烧录实验板的CN3连接器即可。

若要烧录程序，则需使用连接线连接计算机的打印机输出端口与89S51在线烧录实验板的CN1连接器。

除了实体连接外，还需要安装驱动程序。第一次使用时，先安装驱动程序，安装完成后，双击驱动程序的快捷方式，打开如图A-45所示的对话框。



图A-45　s51_pgm窗口

对此对话框中的项目说明如下。

菜单栏

和其他软件的窗口类似，在窗口上方为菜单栏，包括文件、测试、读取、写入、装置与说明菜单。

- 文件菜单提供5个命令，加载程序命令的功能是加载所要烧录的文件（*.hex文件），Reload命令的功能是重新加载文件，另存新文件命令的功能是将目前的数据另存为新文件，储存命令的功能是将目前的数据保存，离开命令的功能是关闭程序。
- 测试菜单提供设定所要写入芯片的测试值。
- 读取菜单提供3个命令，读取命令的功能是读取芯片内容，比对命令的功能是比对烧录结果，空白检查命令的功能是检查芯片是否空白。
- 写入菜单提供3个命令，烧录命令的功能是进行烧录，芯片抹清除命令清除芯片内容，写入命令的功能是进行写入芯片，其中的烧录命令包含清除与写入两个动作。
- 装置菜单提供设定与89S51在线烧录实验板连接口的命令。
- 说明菜单提供版本说明，打开本菜单后，屏幕上出现如图A-46所示的版本说明对话框，只要单击 确 定 按钮即可关闭。



图A-46 版本说明对话框

工作字段

窗口中间是所要操作的数据，当加载所要写入芯片的hex文件或读取芯片的数据时，数据将会显示在此区域之中。除了中间区域外，还有如下三个字段。

- 状态区显示目前芯片的状态，若显示备妥表示目前芯片已待数据写入；若芯片正在进行烧录或比对，则显示红色的忙碌。若进入数据写入芯片或比对数据时，在状态区右边有个进度条，以图形的方式来表示数据写入芯片或比对数据的进度。
- 检误码栏中显示的是检误码字段（Checksum）。
- 窗口下方为记录字段，所有操作状态将记录在此字段中。

操作按钮

程序提供了用于执行常用操作的按钮，而不必使用菜单里的命令，对这些按钮与选项的说明如下。

- **VDD**选项用于设定烧录后是否将电源加到芯片上。若选中本选项，则烧录完毕后立即执行其中的程序；若烧录完成后想要立即拔取89S51，则不要选中本选项。
- 晶片重置 按钮的功能是将芯片复位，相当于按实验板上的Reset按钮。
- 载入程式 按钮的功能是加载所要写入芯片的程序（*.hex），单击本按钮后，即可在随即出现的对话框里指定所要烧录的程序。

- 范例程式 按钮的功能是加载程序所预置的变形霹雳灯程序，如此一来，即使没有准备好自己设计的程序，也能进行89S51在线烧录实验板的测试。单击本按钮后，再单击 烧录 按钮，即可将预置的程序烧录到芯片。若选中了**VDD**选项，则完成烧录后立即执行程序。
- 晶片偵測 按钮的功能是检测目前在试验板上的芯片，单击本按钮后，程序即检测芯片，检测结果将反应在右下方的芯片选择字段，并记录在记录字段里。
- 烧录 按钮的功能是将加载的文件烧录到芯片，其中包括3个步骤：芯片清除、写入和比对，完成烧录后，屏幕上将出现如图A-47所示的对话框，单击 确定 按钮关闭即可。



图A-47　完成烧录

- 晶片清除 按钮的功能是清除芯片的内容，使其变成空白。
- 空白檢查 按钮的功能是检查芯片是否为空白（没有程序），完成检查后，将出现如图A-48所示的对话框，以显示检查结果是否为空白。



图A-48　完成空白检查

- 读取 按钮的功能是读取芯片内的程序，单击本按钮后，即读取芯片内的程序。
- 写入 按钮的功能是将数据写入芯片。
- 比对 按钮的功能是将芯片内的数据与加载的程序比对。单击本按钮后即进行比对，屏幕上将出现如图A-49所示的对话框，单击 确定 按钮关闭即可。



图A-49　完成比对

- 保护 按钮的功能是设定保护功能，单击本按钮后，将选中下面保护位区域中的3个选项，进行烧录时，将依据这3个选项设定是否将该保护位烧掉，以防止芯片中的程序被读取。不过在设定保护之前，一定要慎重考虑，因为保护位烧掉后，不能恢复！
- 芯片选择字段的功能是指定所要烧录的芯片种类。
- 清除记录 按钮的功能是清除左边记录区域的数据。

虽然程序提供很多功能，实际上的操作却很简单！首先利用打印机连接线把计算机的打印机端口与89S51在线烧录实验板的CN1连接起来；再用USB数据线把计算机的打印机端口与89S51在线烧录实验板的CN2连接起来。最后，将芯片放入脚座后即可运行s51_pgm程序，屏幕上将出现如图A-45所示的对话框，接下来对操作进行说明。

测试

若要测试89S51在线烧录实验板是否正常，则单击[范例程式]按钮加载范例程序，再单击[烧 录]按钮即可进行烧录。完成烧录后，屏幕上出现如图A-47所示的对话框，单击[确 定]按钮关闭即可。若选中了**VDD**选项，则完成烧录后即执行程序，也就是先单灯左移，移到最左边后，再从最左边的LED逐一点亮到最右边，如此循环。

烧录

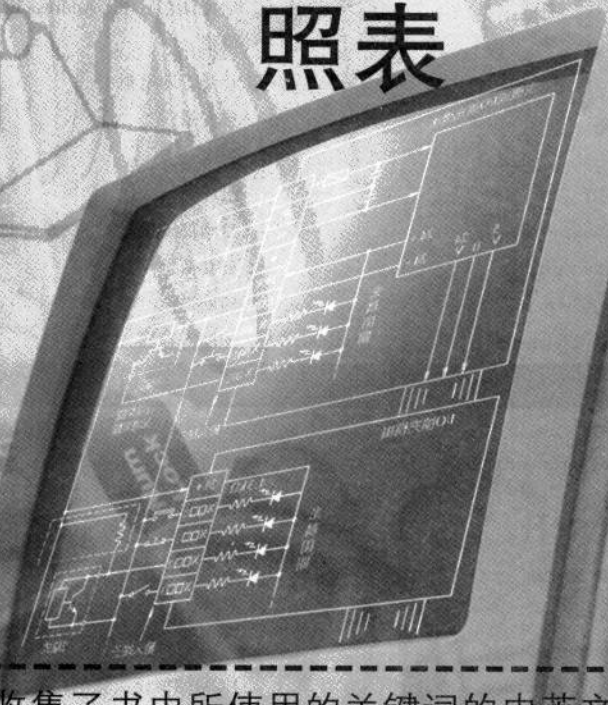
89S51在线烧录实验板的主要功能就是将已设计的程序写入芯片。当需要进行烧录时，首先单击[载入程式]按钮，然后在随即出现的对话框里指定所要加载的文档，再单击[打开(O)]按钮将它载入。然后，再单击[烧 录]按钮，即可进行烧录。完成烧录后，屏幕上出现如图A-47所示的对话框，单击[确 定]按钮关闭即可。若选中了**VDD**选项，则完成烧录后即执行程序。

编辑器

编译器

附录B

中英文名词对照表



本附录收集了书中所使用的关键词的中英文对照，以帮助使用者快速查阅。

A

absolute temperature scale	绝对温度
Accumulator	累加器
ADC	模拟-数字转换器，Analog to Digital Converter的简称
address	地址
Address Counter	LCM的地址计数器，简称为AC
address byte	地址字节
advanced GDI driver	高级图形接口驱动程序
ALE	地址锁存启用，Address Latch Enable的简称
Analog	模拟
armature	转子，即电枢
arguments	自变量
array	数组
Assembler	组译器
Auto-Reload Mode	自动加载模式
Automatic variables	自动变量
Auxiliary Carry	辅助进位
Auxiliary Register	辅助寄存器

B

baud rate	比特率，即串行口的传输率
baud rate generator mode	比特率产生器模式
beat	节拍
BCD	十进制BCD码，即Binary-Coded Decimal的简称
BCD DIP Switch	十进制指拨开关
blanking in / ripple-blanking output	遮没输入或涟波遮没输出引脚
bias voltage generator	LCM的偏压产生电路
bit	位
bouncer	抖动
bps	每秒传输多少位，即bit per second的简称
break	中止
Busy Flag	LCM的忙碌标志位，简称BF
button	按钮开关
buzzer	蜂鸣器
byte	字节

C

CAN	微处理电路内的局域网络控制器，即Controller Area Network的简称
capture mode	捕捉模式

capture register	捕捉寄存器
Celsius temperature scale	摄氏温度
carry	进位
CG RAM	LCM的自建字型产生器，即Character Generate RAM 的简称
CG ROM	LCM的内含字型产生器，即Character Generate ROM 的简称
character	字符，简称为char
clean display	LCM的清除显示屏
closed loop	闭回路
column	行
comment	批注
Common Anode	共阳极，简称为CA
Common Cathode	共阴极，简称为CC或CK
common driver	LCM的共同端驱动电路
compact	精简模式
Compiler	编译器
contant	常量
continuons counting ADC	连续计数式模拟-数字转换器
copy	复制
counter	计数器
CPU	中央处理器，即Central Processing Unit的简称
crystal	石英振荡晶体，简称XTAL
cursor/blink controller	LCM的光标闪烁控制电路
cut	剪切

D

DAC	数字-模拟转换器，Digital to Analog Controller的简称
Darlington Complementary Silicon Power Transistors	互补达林顿功率晶体管
Darlington transistor	达林顿晶体管
DD RAM	LCM的数据显示存储器，即Data Display RAM的简称
data	资料
Data Available	容许数据输出，简称为DA
data base contents	数据库目录
Data Register	LCM的数据寄存器，简称为DR
data sheet	资料单
debouncer	防抖动电路
debug	调试
debug channel	调试通道
Debugger	调试器
decimal point	小数点，简称为dp
default	预置

define	定义
delay	延迟
destination operand	目的操作数
digit	位数
Digital	数位
DIP Switch	指拨开关
direct addressing	直接寻址
disaessembly	反组译
disable	停用
DPTR	数据指针寄存器，Data Pointer register的简称
dual slope ADC	双斜率式模拟-数字转换器
E	
EA	存取外部存储器启用，External Access Enable的简称
Editor	编辑器
EEPROM	可以通过电气来清除与写入的只读存储器，Electrically Erasable Programmable ROM的简称
embedded system	嵌入式系统
enable	启用
enclosure	机壳
EPROM	可以通过电气写入，但需以紫外线清除的只读存储器，Electrically Programmable ROM的简称
evaluation version	评估版
execution	执行
F	
fatch	读取
FILO	先进后出，First In Last Out的简称
flag	标志位
float	浮点数
flow chart	流程图
FPGA	场效可编程门阵列，Field Programming Gate Array的简称
function	函数
G	
global variables	整体变量
Graphic display RAM	显示图案存储器，简称为GDRAM
H	
half-height characters	半高字型
I	
ICE	在线仿真器，In-Circuit Emulator的简称
icon	图标

icon RAM	自建图标产生器
IDE	整合开发环境，Integrated Development Environment的简称
idle mode	闲置模式，简称为IDL模式
IE	中断启用寄存器，Interrupt Enable register的简称
immediate addressing	立即寻址
include	包含
index addressing	索引寻址
index register	索引寄存器
indirect addressing	间接寻址
Instruction Register	LCM的指令寄存器，简称为IR
interrupt	中断
interrupt system	中断系统
IP	中断优先级寄存器，Interrupt Priority register的简称
ISP	在线烧录，In-System Programmable的简称
J	
jumper	跳线
K	
Kelvin temperature scale	开氏温度
keyboard	键盘
Keyboard Mask	按键遮没信号（引脚），简称为KBM
keypad	键盘
Knife Switch	单刀开关或门刀开关，简称为KS
L	
label	标签字段
Lamp Test	测试灯的功能，简称LT
large	大型模式
LCD	液晶显示器，即liquid crystal display的简称
LCD panel	LCD面板
LCM	将LCD面板、驱动与控制电路组合而成的LCD模块，即liquid crystal display moulde的简称
leading zero suppression	消除整数左边（前置）的0
LED	发光二极管，Light-Emitting Diode的简称
link	链接
Linker	链接器
local variables	区域变量
Locator	链接器
logic analyzer	逻辑分析仪
LSB	最低有效位（最右边位），即为Least Significant Bit简称
LSD	最低有效位（最右边位），即为Least Significant Digit简称

M	
machine cycle	机器周期
Mask	光罩
Mask ROM	光罩式只读存储器
master processor	主处理器
MCS	微控制系统Micro Controller System的简称
memory	存储器
memory models	存储器模式
mnemonic	指令助记码字段
module	模块
MSB	最高有效位（最左边位），即为Most Significant Bit简称
MSD	最高有效位（最左边位），即为Most Significant Digit简称
N	
noise margin	噪声容限
normal closed	常闭接点，简称NC接点
normal opened	常开接点，简称NO接点
O	
object code	目标码
On-chip Emulator Logic	芯片内置仿真器逻辑
Open Drain	漏极开路输出，简称为OD
operand	操作数
operator	运算符
Oscillator	振荡器，简称OSC
Output Enable	输出启用，简称为OE
overflow	溢出
P	
parallel-serial converter	LCM的并行/串行数据转换器
parallel port	并列口
parity bit	串行数据奇偶位
Parity check	奇偶检查
paste	粘贴
PC	程序计数器，Program Counter的简称
PCON	电源控制寄存器，Power Control register的简称
PDIP	双并排双列直插式封装，Plastic Dual Inline Package的简称
peripherals	外围装置
PLCC	四方形J型引脚IC封装，Plastic J-Leaded Chip Carrier的简称
pointer	指标
POR	开机复位，Power On Reset的简称
port	输入/输出口

power-down mode	功率下降模式，简称为PD模式
PQFP	四方形扁平表面贴式L型引脚IC封装，Thin Plastic Gull Wing Quad Flatpack的简称
prefermace analyzer	效能分析器
Pre-Processor	前置处理器
project	专案或专题
Project Manager	项目管理器
Prototype	原型板或原型机
PSEN	程序储存启用，Program Store Enable的简称
PSW	程序状态字组，Program Status Word的简称
PWM	脉冲宽度调变，pulse width modulation的简称
PDM	脉冲宽度调变，pulse duration modulation的简称，同PWM
R	
RAM	随机存取存储器，Randon Acess Memory的简称
register	寄存器
register addressing	寄存器寻址
Register Bank	寄存器组
register bank selector	寄存器组选择位
register selector	LCM内部寄存器选择引脚
relay	继电器
reset	复位
results	结果
Ripple-Blanking Input	涟波遮没输入引脚，简称为RBI
ROM	只读存储器，Read Only Memory的简称
ROM less	无程序存储器
row	列
RTOS	实时操作系统，Real Time Operation System的简称
S	
SBUF	串行口缓冲器，Serial Buffer的简称
SCON	串行端口控制寄存器，Serial Port Control register的简称
segment dirver	LCM的区段驱动电路
Serial Peripheral Interface Bus	串行外围装置接口总线，即SPI总线
SFR	特殊功能寄存器，Special Function Register的简称
signed	符号数字
simulation	仿真
slave processor	从处理器
small	小型模式
SOC	开始转换，start of convert的简称
source code	源程序
source operand	源操作数

SPI	串行外围装置接口，即Serial Peripheral Interface的简称
speaker	喇叭
Stack Pointer register	堆栈指针寄存器
start bit	串行数据的起始位
stator	定子
stepping motor	步进马达
stop bit	串行数据的停止位
successive-approximation ADC	连续接近式模拟-数字转换器
symbol analyzer	符号分析器
synchronous serial data link standard	同步串行数据连接标准
T	
Tack Switch	按钮开关
target board	目标板
TCON	计时/计数器控制寄存器，Timer/Counter Control register的简称
Timer	定时器
timing generator	LCM的时序产生电路
TMOD	计时/计数模式控制寄存器，Timer/Counter Mode Control register的简称
toggle	切换
Tone	音调
Toolbox	工具箱
TQFP	四方形扁平表面贴式IC封装，Thin Plastic Gull Wing Quad Flatpack的简称
trailing zero suppression	消除小数右边（尾端）的0
TTL	晶体管-晶体管逻辑，Transistor-Transistor Logic的简称
U	
UART	万用异步串行端口，Universal Asychronuse Receiver-Transmitter的简称
unsigned	无符号数字
V	
variables	变量
void	无
W	
watch	监控
WDT	看门狗定时器，Watchdog Timer的简称
WDTRST	看门狗定时器复位寄存器，Watchdog Timer Reset register的简称